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

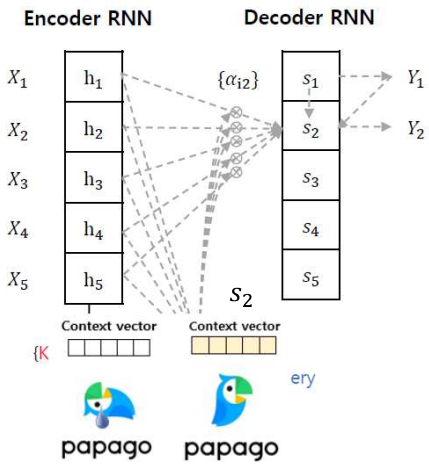
- Attention: Query와 key의 Similarity를 계산한 후 value 의 가중합을 계산
- Attention score: Value 에 곱해지는 가중치
- Considerations
  - ✓ Key, Query, Value = Vectors (Matrix/Tensor)
  - ✓ Similarity function

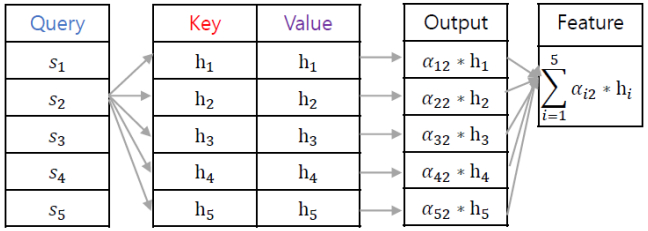
$$A(q,K,V) = \sum_i softmax\big(f(K,q)\big)V$$

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



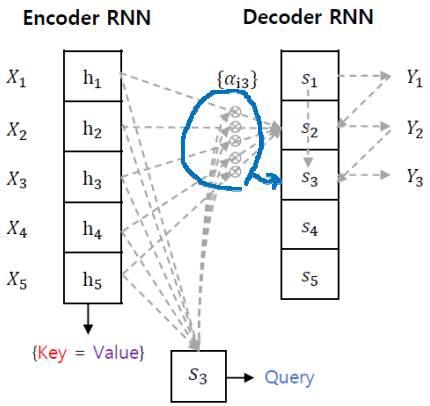


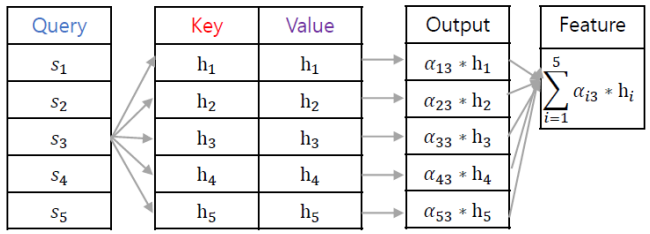
$$\{\alpha_{i2}\} = softmax(f(h_i, s_2))$$

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

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

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

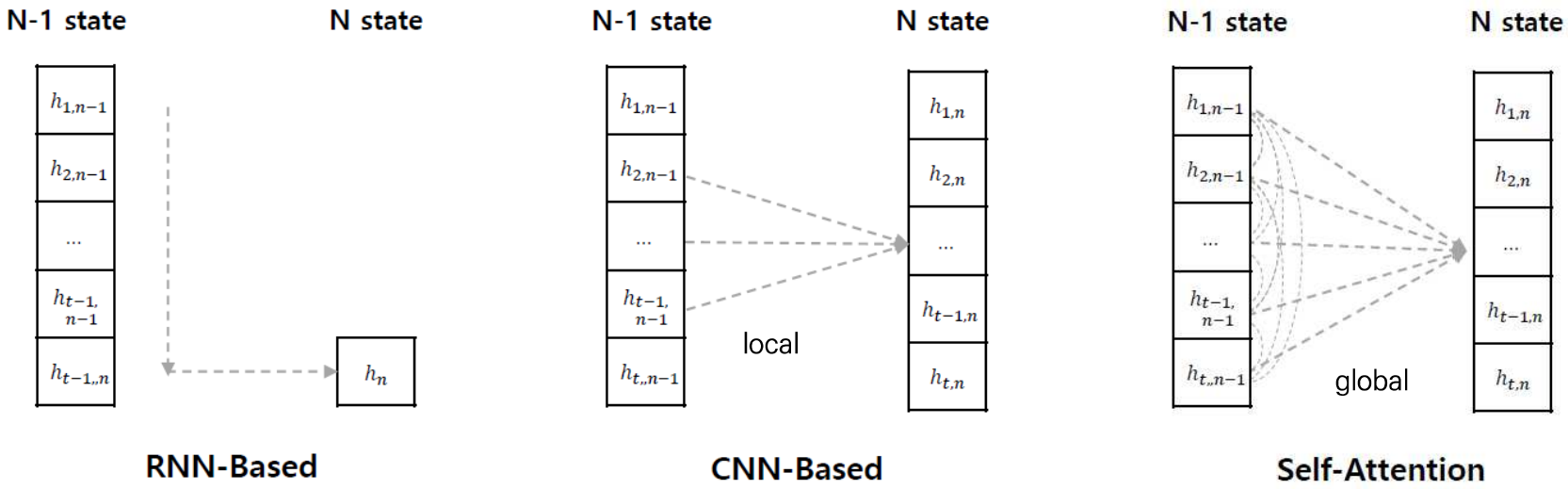
Self-Attention

➢ 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

$Key = Query = Value$

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



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

✓ Long path length between long-range dependencies

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

✓ 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

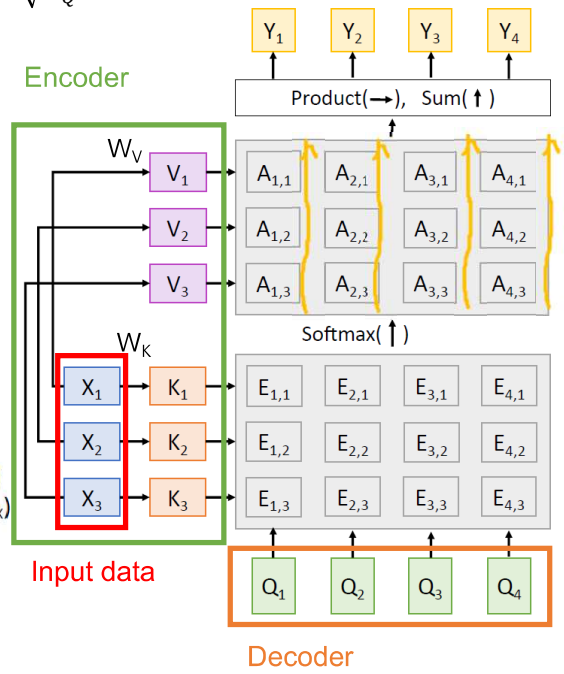
- Attention (Decoder → Query / Encoder → Key, Value) : Encoder, Decoder 사이의 상관관계를 바탕으로 특징 추출
- Self attention (입력 데이터 → Query, Key, Value) : 데이터 내의 상관관계를 바탕으로 특징 추출

$$Y_i = \sum_j softmax(\frac{Q_i(X_jW_K)^T}{\sqrt{D_Q}})X_jW_V$$

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

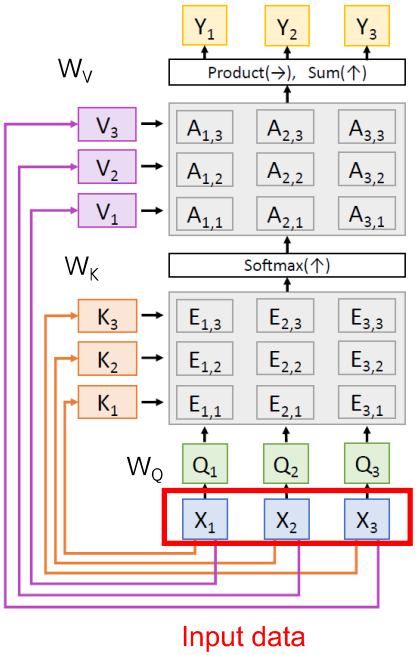
$$Y_i = \sum_j softmax(\frac{X_iW_Q(X_jW_K)^T}{\sqrt{D_Q}})X_jW_V$$

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)

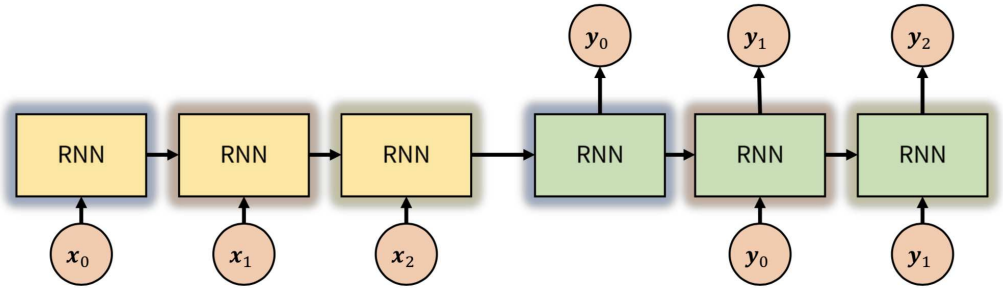
Computation:
Query vectors: $Q = XW_Q$
Key vectors: $K = XW_K$ (Shape: N_X x D_Q)
Value Vectors: $V = XW_V$ (Shape: N_X x D_V)
Similarities: $E = QK^T$ (Shape: N_X x N_X) $E_{i,j} = Q_i \cdot K_j / \sqrt{D_Q}$
Attention weights: A = softmax(E, dim=1) (Shape: N_X x N_X)
Output vectors: $Y = AV$ (Shape: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

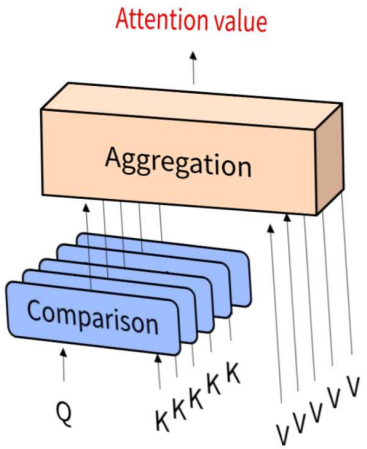
Attention



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.

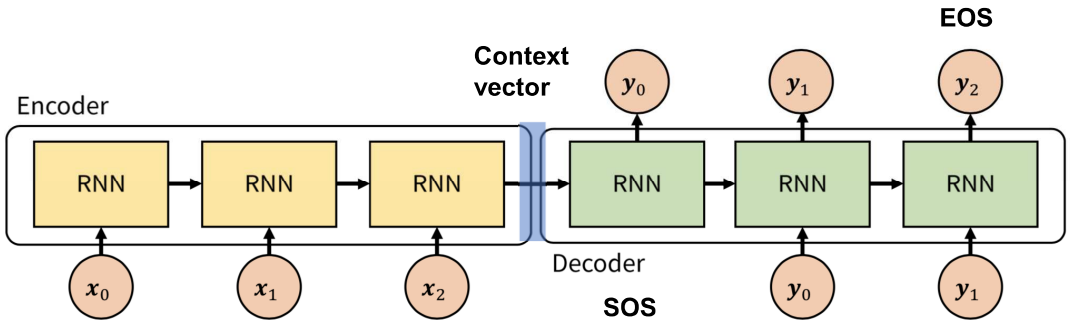
Q에 대해 어떤 K가 유사한지 비교하고, 유사도를 반영하여 V들을 합성한 것이 Attention value이다.

Attention

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

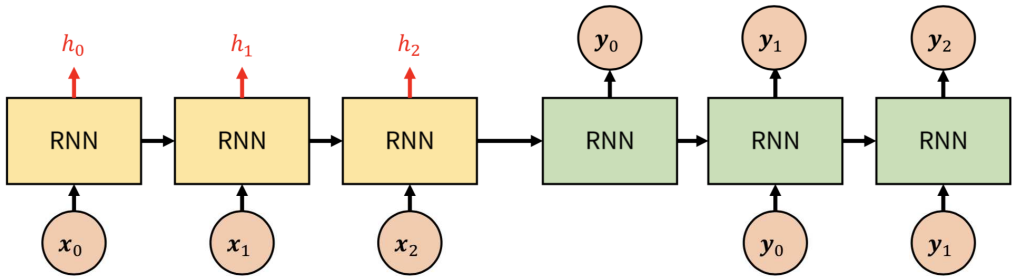
Seq2seq

- 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.

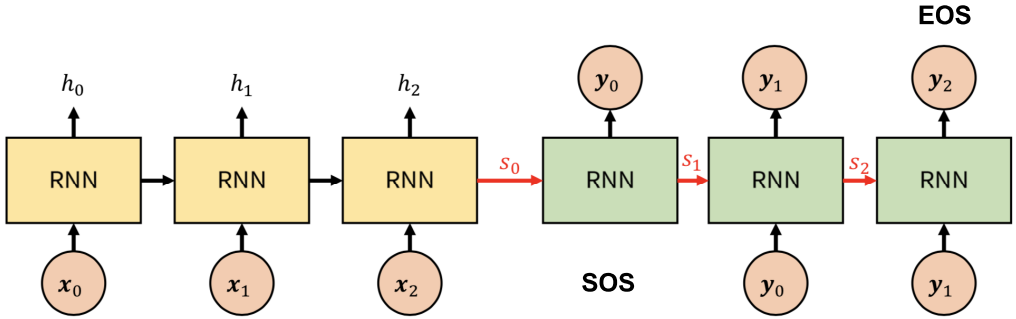
- 현재의 출력을 내기 위해서는 현재의 출력이 사용될 수는 없기 때문에 즉, 미래(예측)를 가지고 현재 출력을 만들어낼 수는 없기 때문에 하나 앞선 Decoder Hidden state s_i를 query로 사용하는 것이다.
- 대부분의 Attention network에서는 key와 value를 같은 값을 사용한다. Seq2seq에서는 Encoder의 Hidden Layer들을 key와 value로 사용한다.

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.



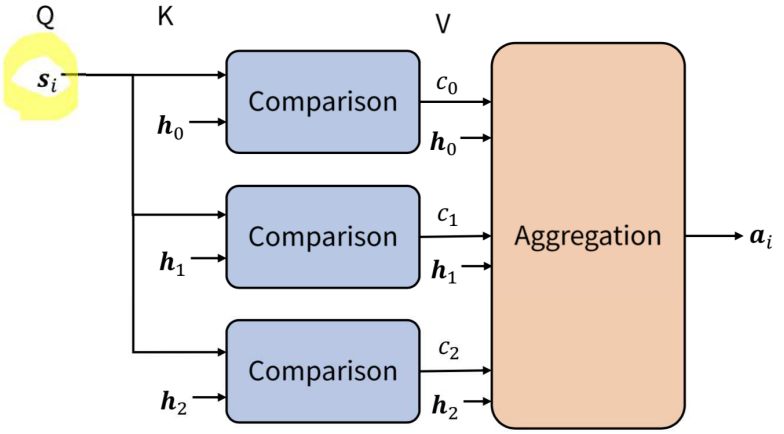
Seq2seq에서는 Decoder의 hidden layer들을 Query로 사용한다.

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.



$$c_i = \text{softmax}(s_i^T h_j)$$

$$a_i = \sum_j c_i h_j$$

i번째 decoder에 대해서 a_i의 attention value를 얻는다.

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

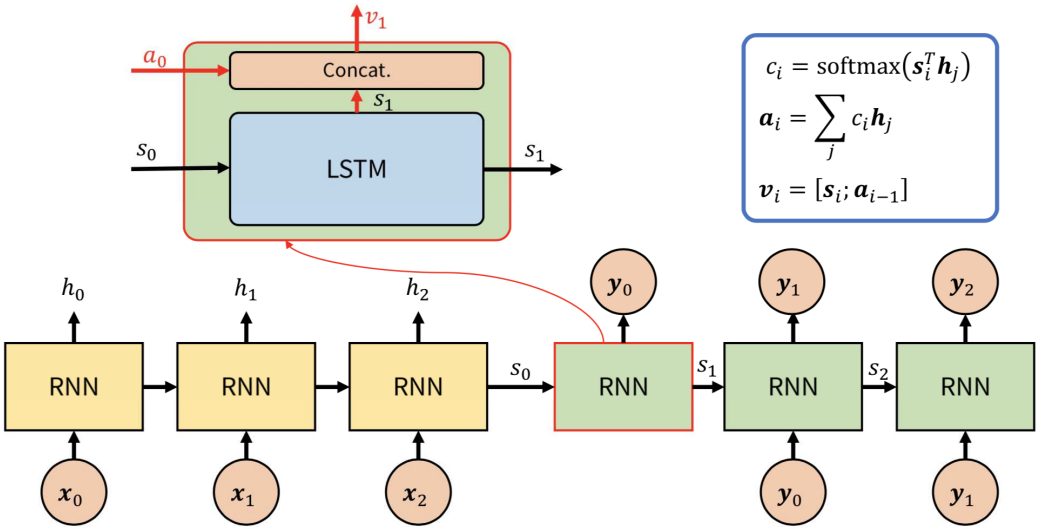
- Key와 Value는 서로 동일한 Encoder의 Hidden State h_i 들이 사용되며, Query는 Decoder에 있는 각각의 Hidden State s_i 들이 될 것이다.
- i-번째 time step에 대한 Query를 보내어 Encoder에 있는 모든 Key와 유사도를 비교해서 최종적으로는 유사도를 고려한 Aggregation한 Attention Value를 출력한다.

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.

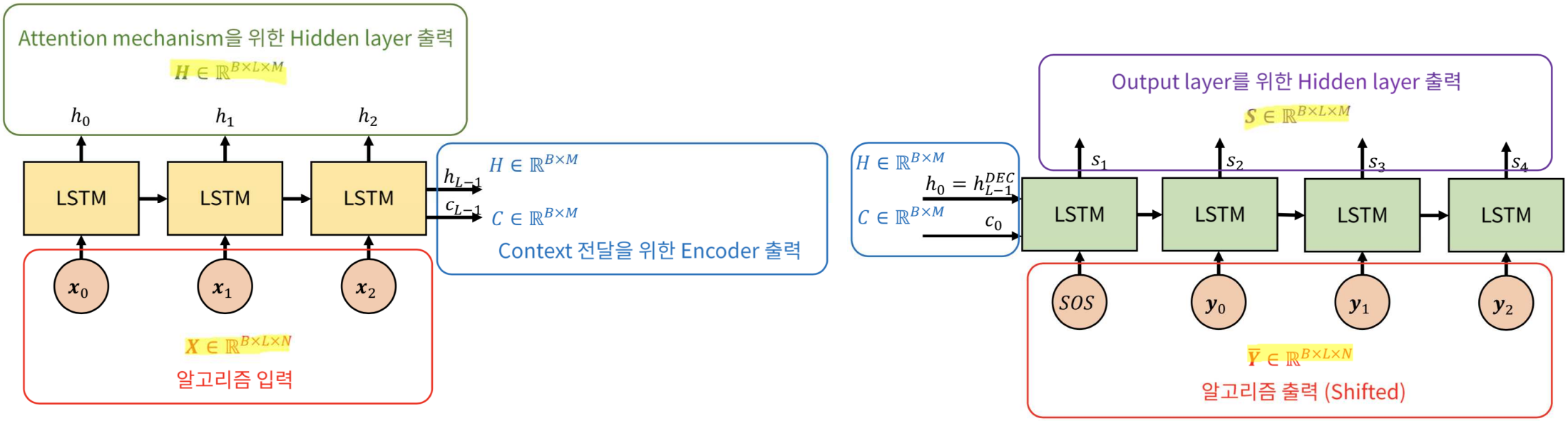
Attention

[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

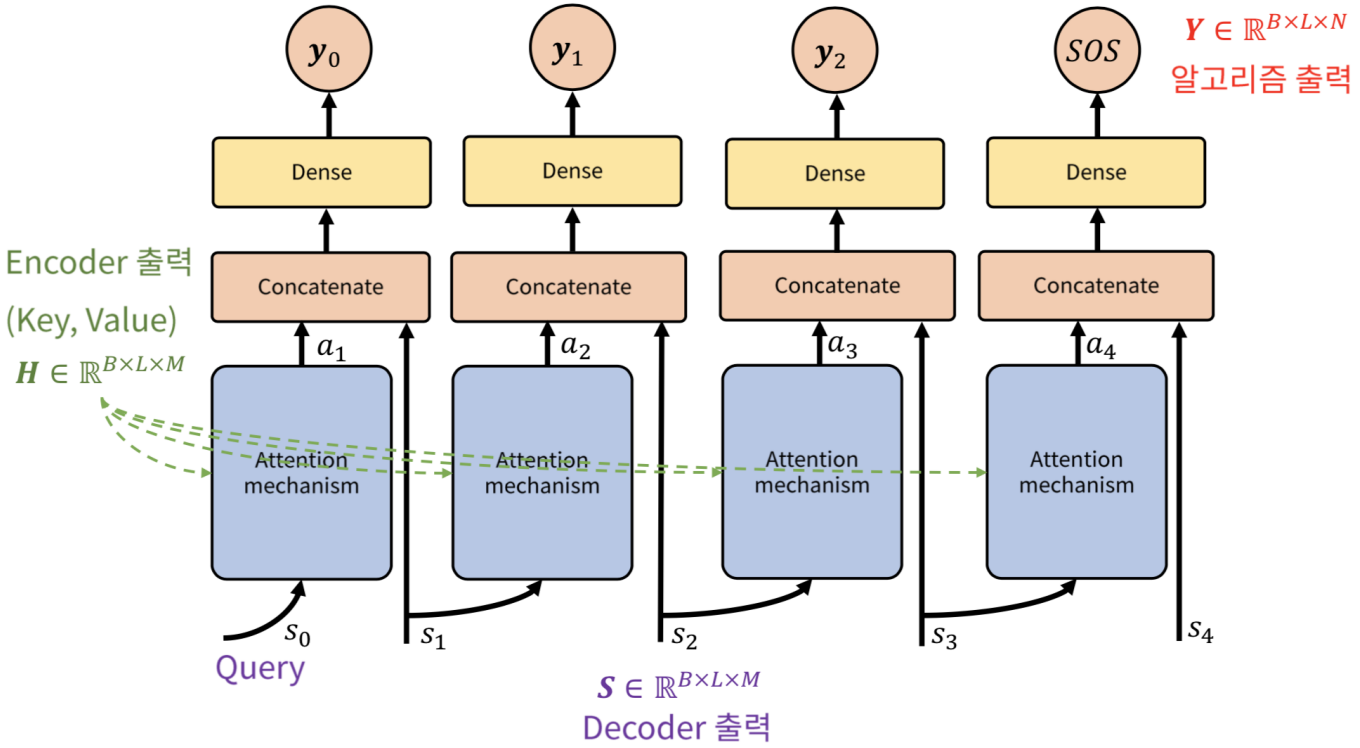


Attention

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

Seq2seq – Output w/Attention (학습단계)

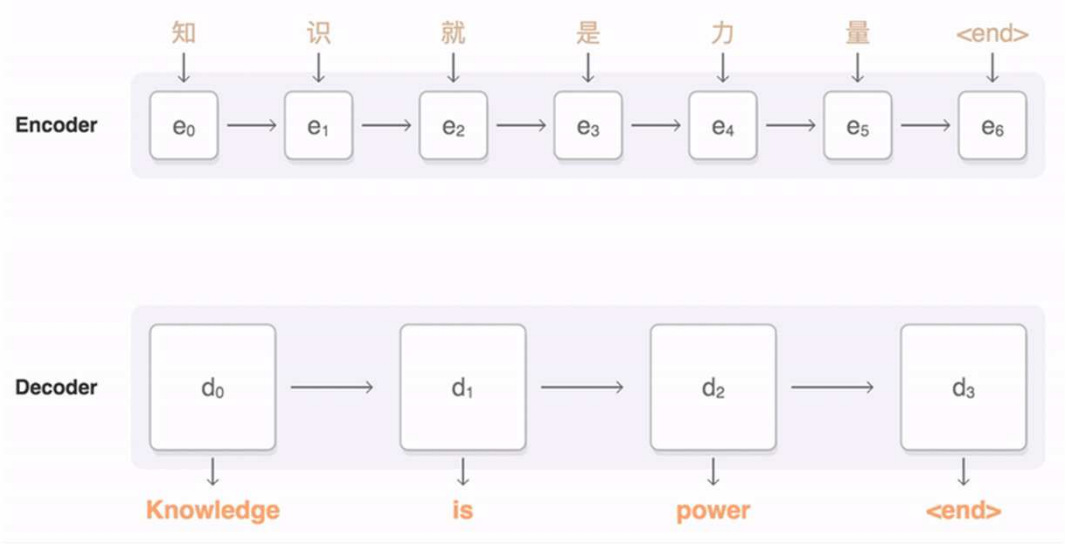
- Encoder의 Hidden State인 H가 Attention mechanism에 Key와 Value로 입력이 되고, Query에는 Decoder의 한 step 앞선 Hidden State를 사용하게 된다.

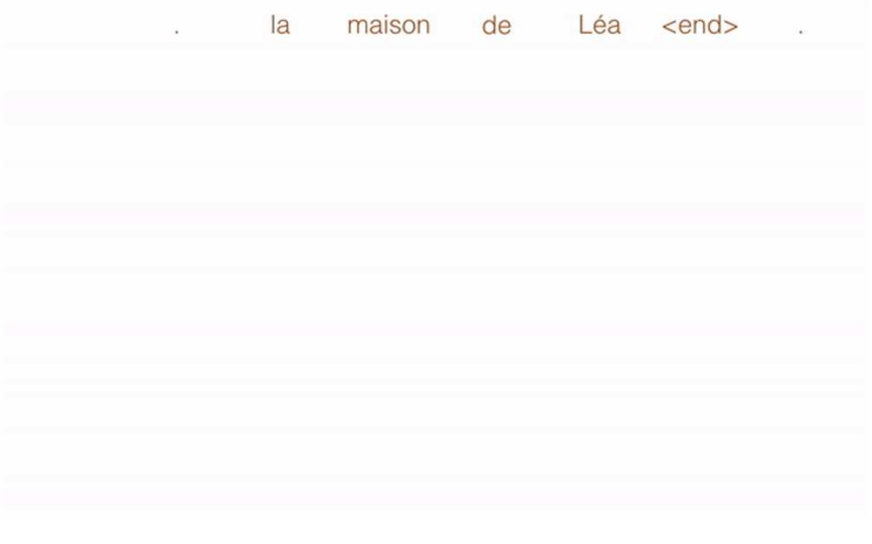


Transformer : Attention is all you need

[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

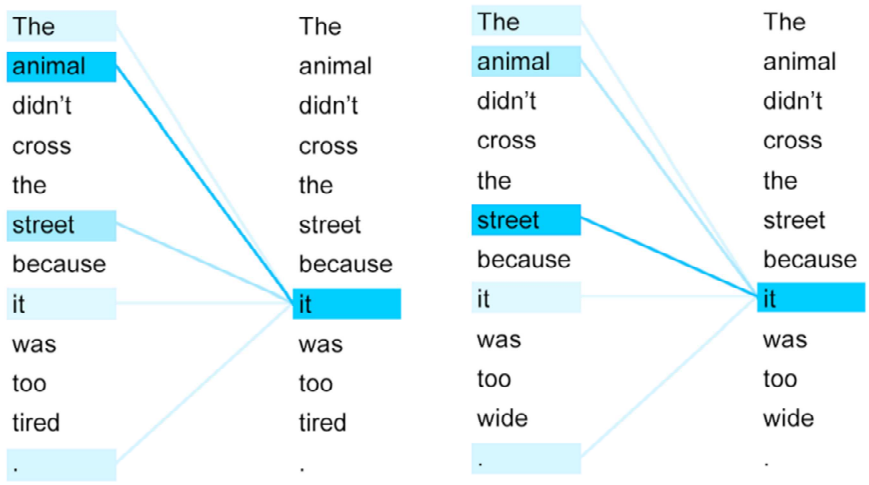
Layer Type	Complexity per Layer	Sequential Operations	Maximum Path Length
Self-Attention	$O(n^2 \cdot d)$	$O(1)$	$O(1)$
Recurrent	$O(n \cdot d^2)$	$O(n)$	$O(n)$
Convolutional	$O(k \cdot n \cdot d^2)$	$O(1)$	$O(log_k(n))$
Self-Attention (restricted)	$O(r \cdot n \cdot d)$	$O(1)$	$O(n/r)$

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

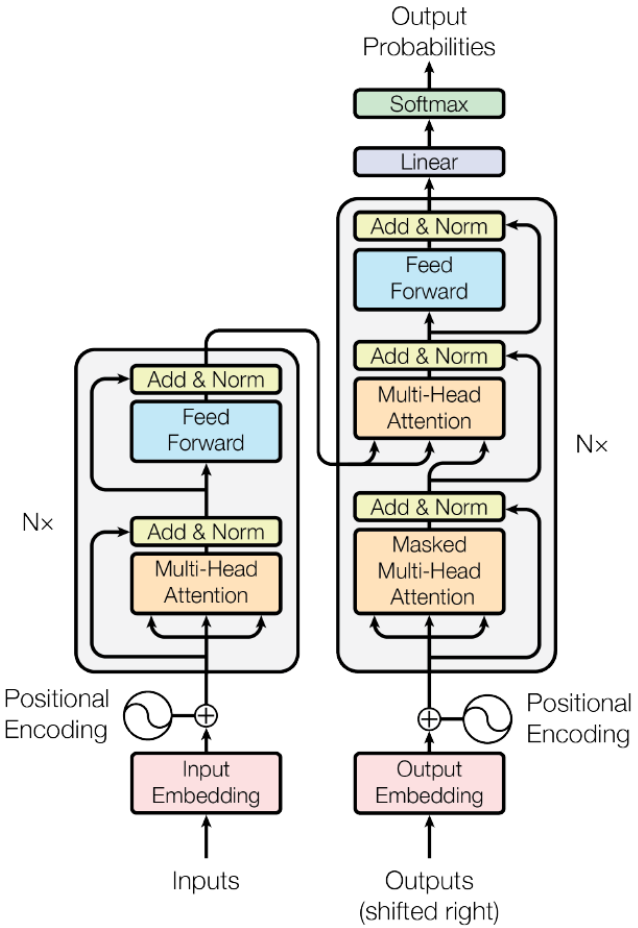


Fig. 1. The Transformer – model architecture

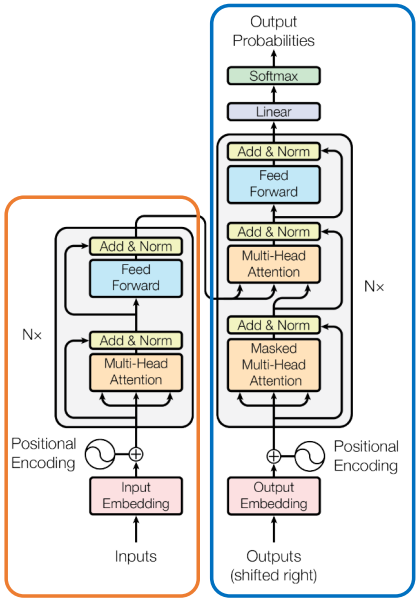
Translation Task에 RNN과 CNN 사용하지 않고, Attention 만을 이용하여 State-of-the-art 성능을 도출한 연구

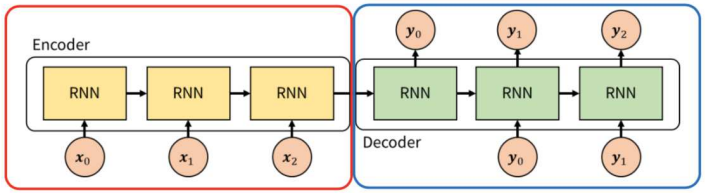
- RNN 같은 경우는 순서대로 입력하기 때문에 입력된 단어의 위치를 따로 표시하지 않아도 되지만, Transformer 구조 같은 경우에는 병렬적으로 계산하므로, 현재 계산하고 있는 단어가 어느 위치에 있는 단어인지를 표현해주어야 해서 *positional encoding*을 사용한다.

- 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

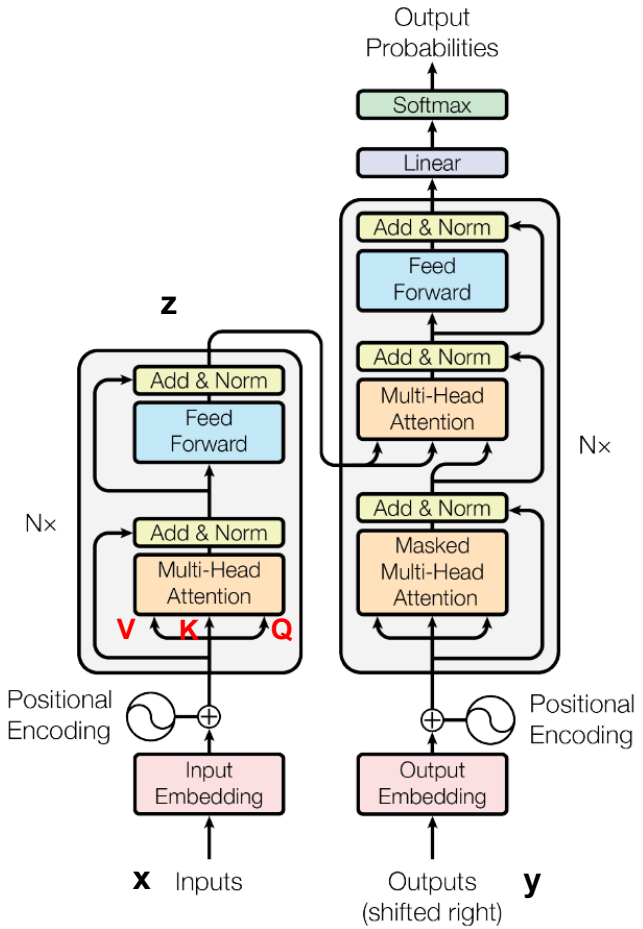


Fig. 1. The Transformer – model architecture

- 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

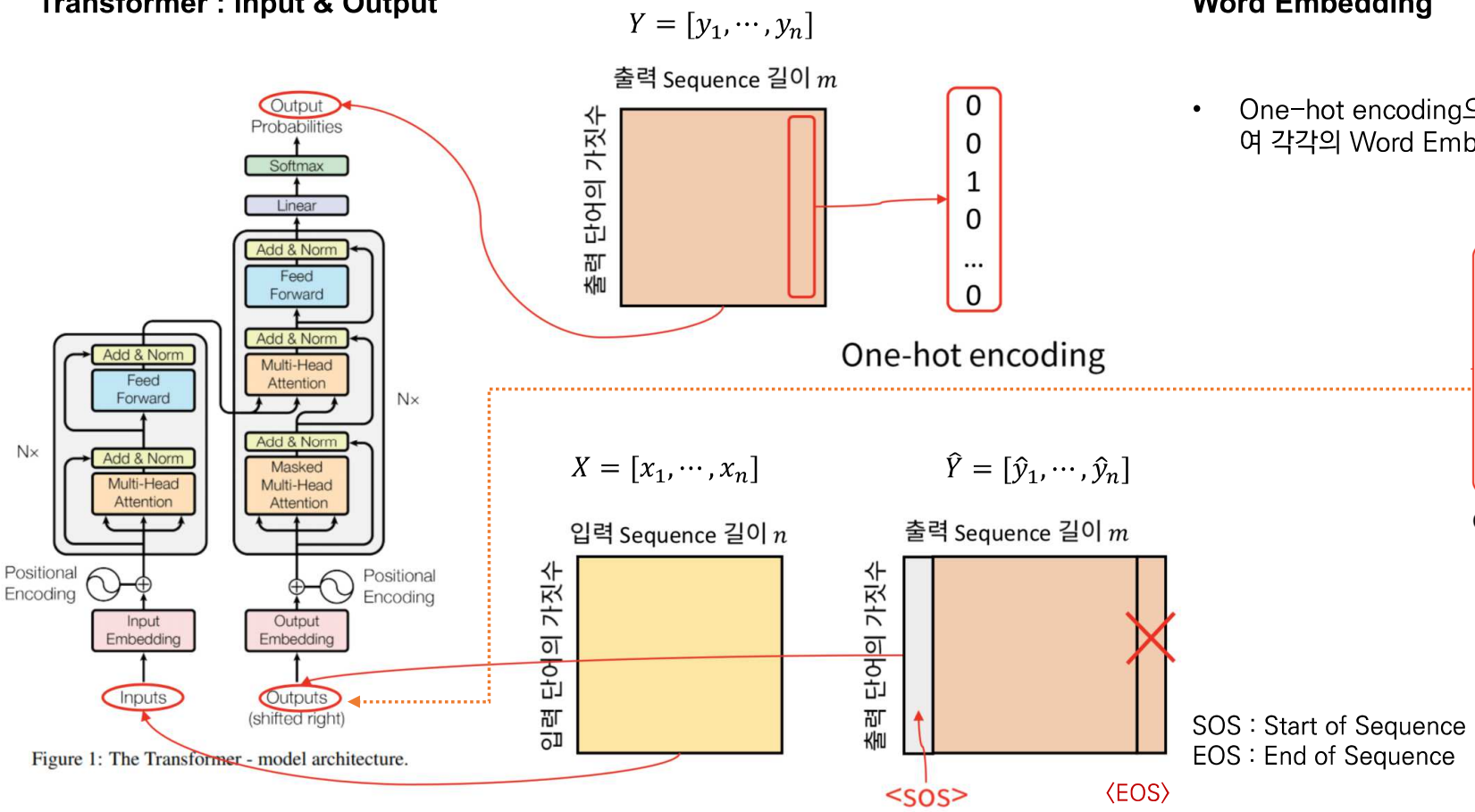
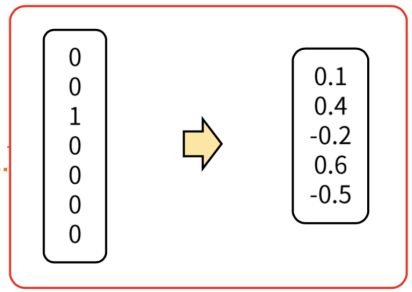


Figure 1: The Transformer - model architecture.

Word Embedding

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



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

SOS : Start of Sequence
EOS : End of Sequence

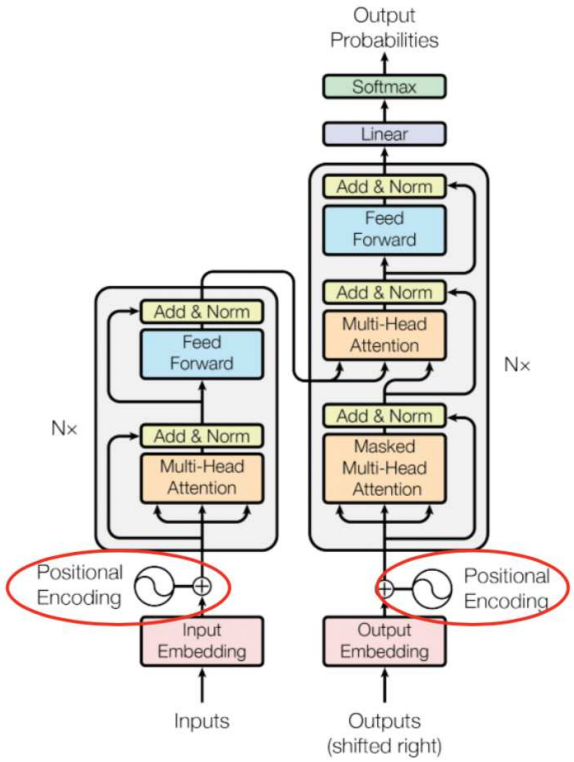
Input : 노란색 Matrix 형태로 되어 있으며, 일반적으로는 입력 단어의 가짓수와 출력 단어의 가짓수는 동일할 것이다. 만약 기계번역처럼 2개 언어가 다르면, 다를 것이다!

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

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

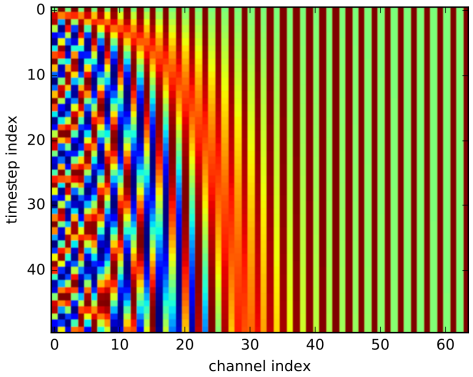


[출처] https://skyjwoo.tistory.com/entry/positional-encoding%EC%9D%B4%EB%9E%80-%EB%AC%B4%EC%97%87%EC%9D%B8%EA%B0%80
https://kazemnejad.com/blog/transformer_architecture_positional_encoding/

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

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

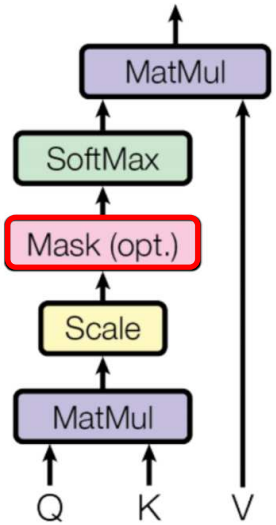
$$PE_{(pos,2i+1)} = \cos(\frac{pos}{10000^{\frac{2i}{d_{model}}}})$$



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

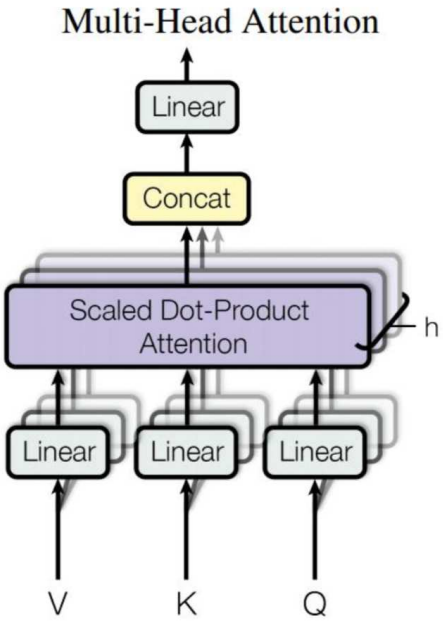
- Small d_k : additive and dot-product attention mechanism perform similarly. Additive attention outperforms dot product attention without scaling for larger d_k.
- Large d_k → dot products grow large in magnitude → pushing the softmax function into regions where it has extremely small gradients.
  → 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**.

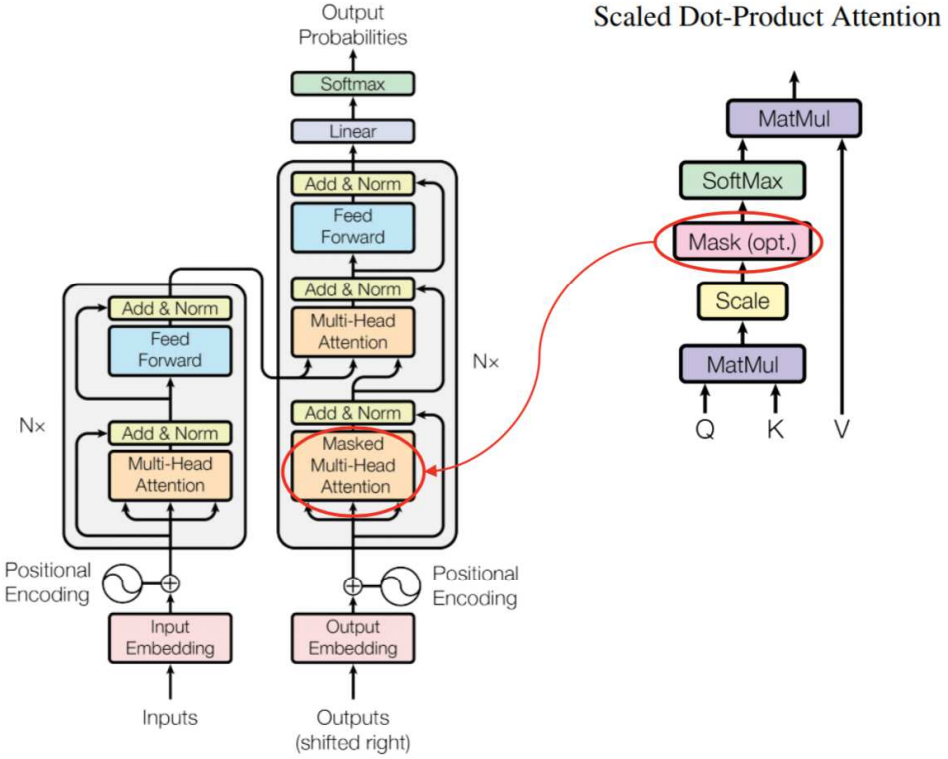
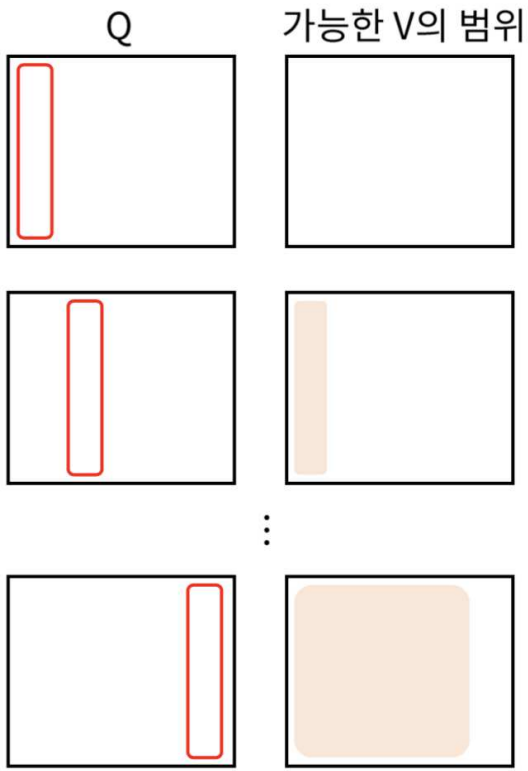


Figure 1: The Transformer - model architecture.



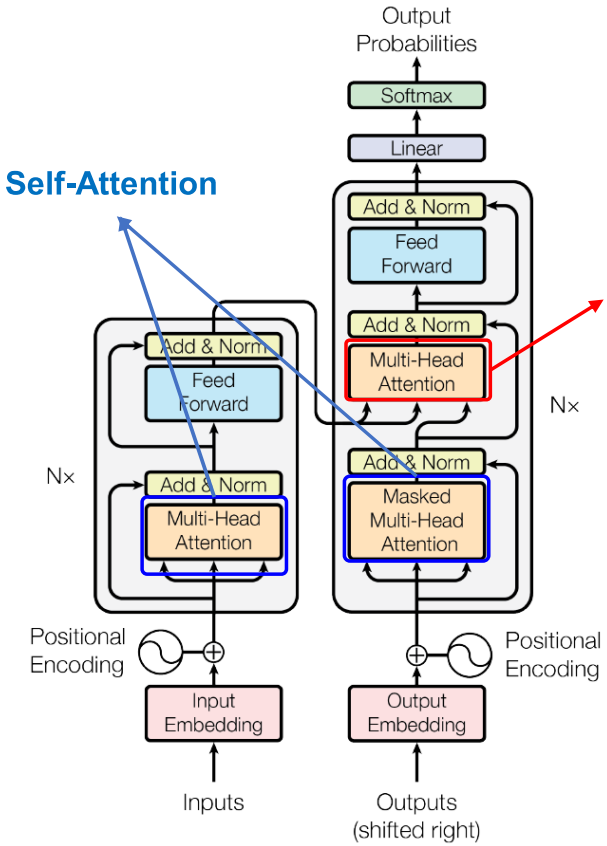
Self-Attention에서 자기 자신을 포함한 미래의 값과는 Attention을 구하지 않기 때문에, Masking을 사용한다.

Transformer : Attention is all you need

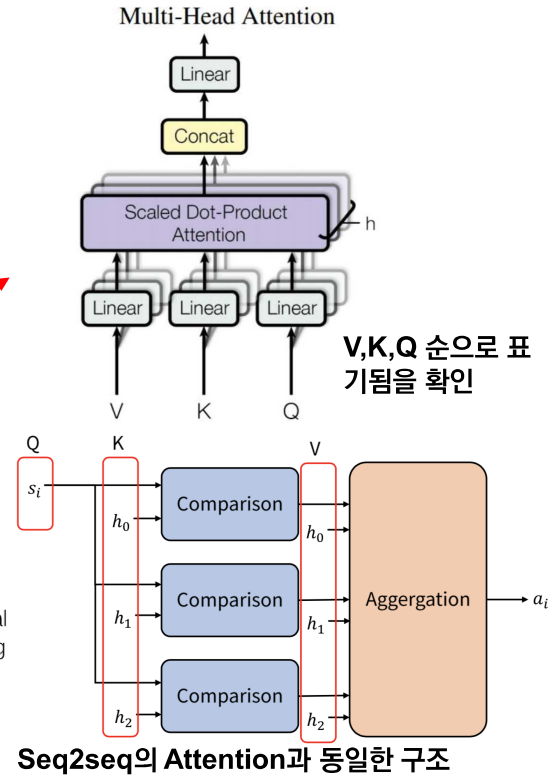
[출처] https://heung-bae-lee.github.io/2020/01/21/deep_learning_10/

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.

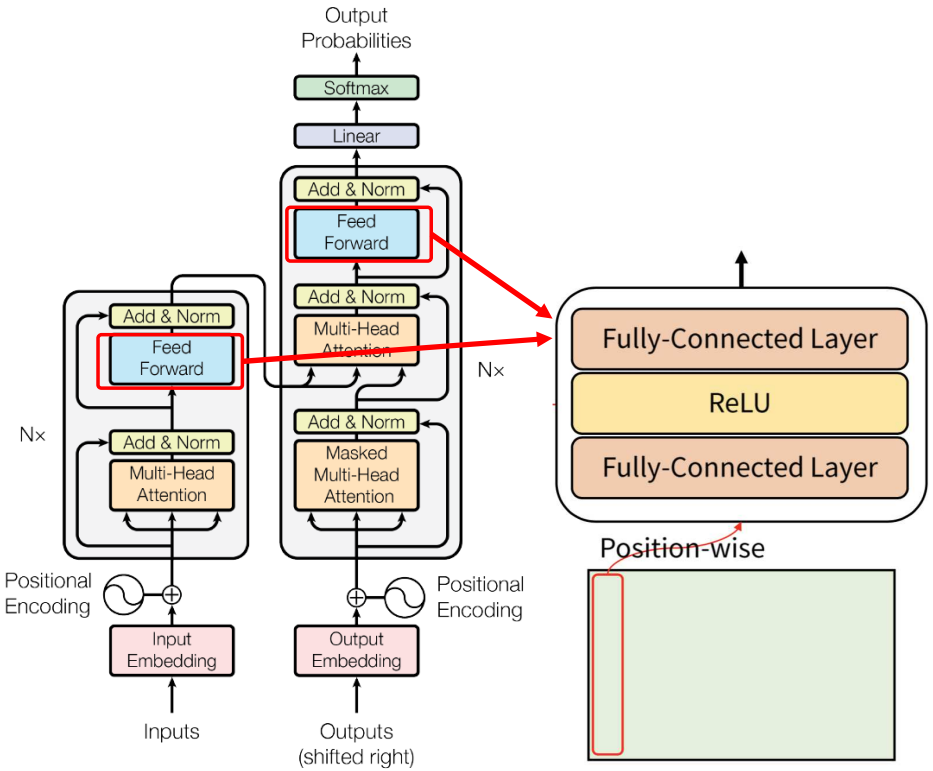
Transformer : Attention is all you need

[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.

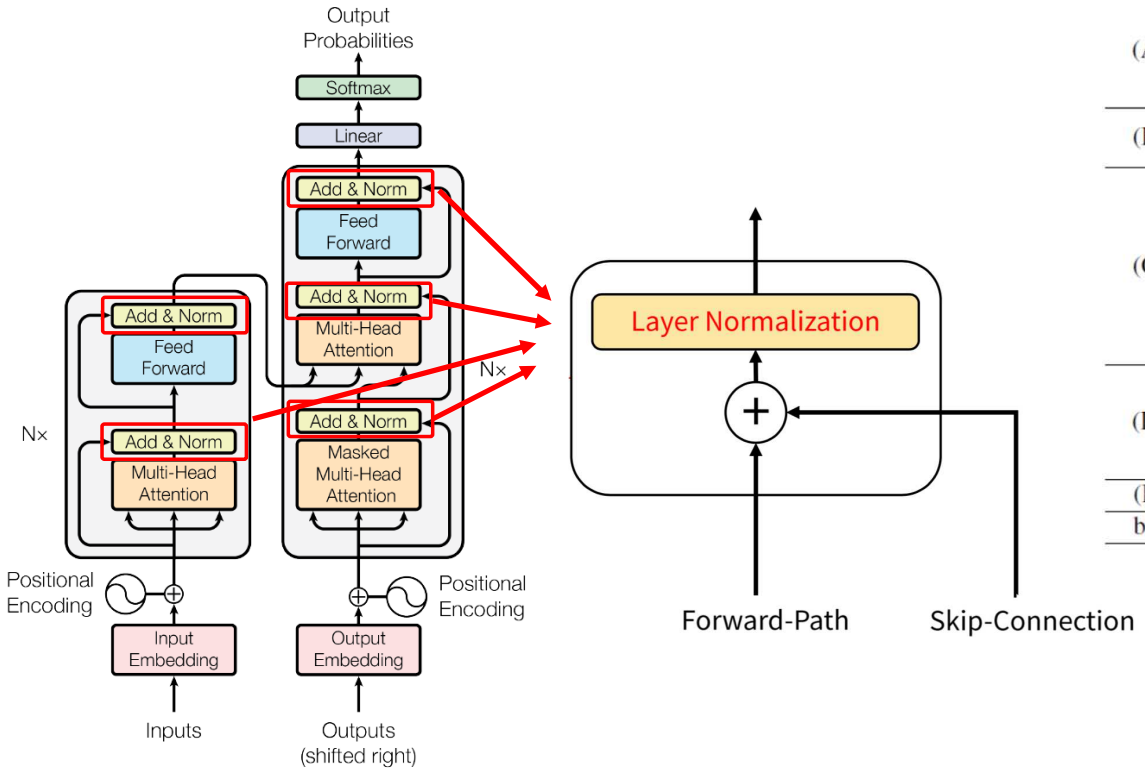


Table 3: Variations on the Transformer architecture. Unlisted values are identical to those of the base model. All metrics are on the English-to-German translation development set, newstest2013. Listed perplexities are per-wordpiece, according to our byte-pair encoding, and should not be compared to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

Transformer 심층분석 PyTorch

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)
```

[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).

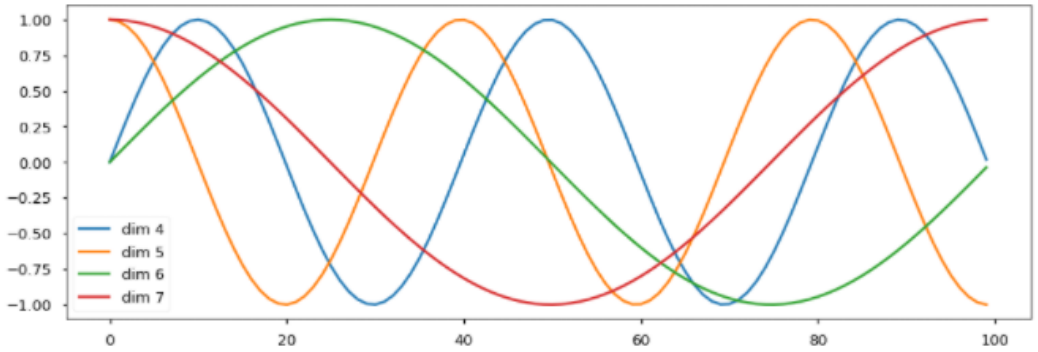
In this work, we use sine and cosine functions of different frequencies:
$PE_{(pos,2i)} = sin(pos/10000^{2i/d_{\text{model}}})$

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

[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 representations을 학습**할 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 정규화**를 포함하고 있다. 그러나 이러한 정규화 유형은 큰 배치 사이즈에 의해 좌우되며, 당연하게도 recurrence에 적합하지 않다. 기존 트랜스포머 아키텍처는 **레이어 정규화**를 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)
```

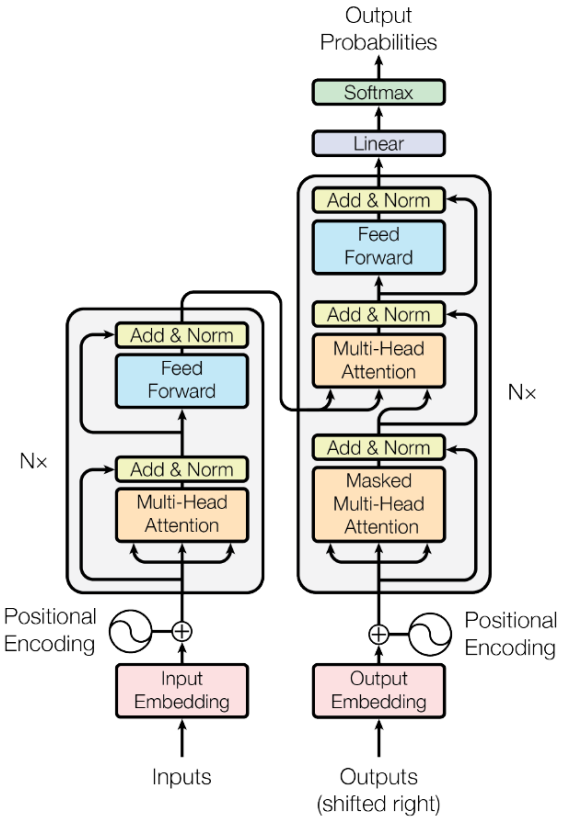
[출처] https://wandb.ai/carlolepelaars/transformer_deep_dive/reports/Transformer-Deep-Dive--VmlldzozODQ4NDQ

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 $\text{matrix } M \in \mathbb{R}^{20\times30000}$이 됨.
- 그런 다음 마지막 차원에 대한 argmax를 취하여 tokenize를 통해 텍스트 열로 디코딩 할 수 있는 출력 토큰 $T \in \mathbb{R}^{20}$ 의 벡터를 얻을 수 있다.

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

```
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

1 ViT (Vision Transformer)

[Google Research, Brain Team]
A. Dosovitskiy, et al. “An Image Is Worth 16X16 Words: Transformers for Image Recognition at Scale,” ICLR2021

We show that

- **The reliance on CNNs is not necessary**
- **Pure transformer applied directly to sequences of image patches** can perform **very well on image classification tasks.**

- When pre-trained on large amounts of data and transferred to multiple mid-sized or small image recognition benchmarks (ImageNet, CIFAR-10, VTAB, etc), **ViT** attains **excellent results compared to SOTA conventional networks** while **requiring substantially fewer computational resources to train.**

google-research/vision_transformer ★ 2,952
official
Quickstart in Colab

huggingface/transformers ★ 47,764 PyTorch

rwightman/pytorch-image-models ★ 11,204 PyTorch
Quickstart in Colab

lucidrains/vit-pytorch ★ 4,799 PyTorch

facebookresearch/vissl ★ 1,742 PyTorch
Quickstart in Colab

See all 50 implementations

	Ours-JFT (ViT-H/14)	Ours-JFT (ViT-L/16)	Ours-I21k (ViT-L/16)	BiT-L (ResNet152x4)	Noisy Student (EfficientNet-L2)
ImageNet	**88.55** ± 0.04	87.76 ± 0.03	85.30 ± 0.02	87.54 ± 0.02	88.4/88.5*
ImageNet ReaL	**90.72** ± 0.05	90.54 ± 0.03	88.62 ± 0.05	90.54	90.55
CIFAR-10	**99.50** ± 0.06	99.42 ± 0.03	99.15 ± 0.03	99.37 ± 0.06	–
CIFAR-100	**94.55** ± 0.04	93.90 ± 0.05	93.25 ± 0.05	93.51 ± 0.08	–
Oxford-IIIT Pets	**97.56** ± 0.03	97.32 ± 0.11	94.67 ± 0.15	96.62 ± 0.23	–
Oxford Flowers-102	99.68 ± 0.02	**99.74** ± 0.00	99.61 ± 0.02	99.63 ± 0.03	–
VTAB (19 tasks)	**77.63** ± 0.23	76.28 ± 0.46	72.72 ± 0.21	76.29 ± 1.70	–
TPUv3-core-days	2.5k	0.68k	0.23k	9.9k	12.3k

Table 2: Comparison with state of the art on popular image classification benchmarks. We report mean and standard deviation of the accuracies, averaged over three fine-tuning runs. Vision Transformer models pre-trained on the JFT-300M dataset outperform ResNet-based baselines on all datasets, while taking substantially less computational resources to pre-train. ViT pre-trained on the smaller public ImageNet-21k dataset performs well too. *Slightly improved 88.5% result reported in Touvron et al. (2020).

1 ViT (Vision Transformer)

[Google Research, Brain Team]
A. Dosovitskiy, et al. “An Image Is Worth 16X16 Words: Transformers for Image Recognition at Scale,” ICLR2021

- Fine-tuning code and pre-trained models are available on https://github.com/google-research/vision_transformer

Vision Transformer and MLP-Mixer Architectures

Update (20.6.2021): Added the "How to train your ViT? ..." paper, and a new Colab to explore the >50k pre-trained and fine-tuned checkpoints mentioned in the paper.

Update (18.6.2021): This repository was rewritten to use Flax Linen API and `ml_collections.ConfigDict` for configuration.

In this repository we release models from the papers

- An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale
- MLP-Mixer: An all-MLP Architecture for Vision
- How to train your ViT? Data, Augmentation, and Regularization in Vision Transformers

The models were pre-trained on the ImageNet and ImageNet-21k datasets. We provide the code for fine-tuning the released models in JAX/Flax.

- Fine-tuning code and pre-trained models are available on https://github.com/jeonsworld/ViT-pytorch

1. Download Pre-trained model (Google's Official Checkpoint)

- Available models: ViT-B_16(**85.8M**), R50+ViT-B_16(**97.96M**), ViT-B_32(**87.5M**), ViT-L_16(**303.4M**), ViT-L_32(**305.5M**), ViT-H_14(**630.8M**)
  - imagenet21k pre-train models
    - ViT-B_16, ViT-B_32, ViT-L_16, ViT-L_32, ViT-H_14
  - imagenet21k pre-train + imagenet2012 fine-tuned models
    - ViT-B_16-224, ViT-B_16, ViT-B_32, ViT-L_16-224, ViT-L_16, ViT-L_32
  - Hybrid Model(Resnet50 + Transformer)
    - R50-ViT-B_16

```
# imagenet21k pre-train
wget https://storage.googleapis.com/vit_models/imagenet21k/{MODEL_NAME}.npz

# imagenet21k pre-train + imagenet2012 fine-tuning
wget https://storage.googleapis.com/vit_models/imagenet21k+imagenet2012/{MODEL_NAME}.npz
```

JAX is Autograd and XLA(Accelerated Linear Algebra), brought together for high-performance numerical computing and machine learning research. It provides composable transformations of Python+NumPy programs: *differentiate*, *vectorize*, *parallelize*, *Just-In-Time compile to GPU/TPU*, and more.

Flax is a neural network library and ecosystem for JAX that is designed for flexibility. Flax is in use by a growing community of researchers and engineers at Google who happily use Flax for their daily research.

1 ViT (Vision Transformer)

A. Dosovitskiy, et al. “An Image Is Worth 16X16 Words: Transformers for Image Recognition at Scale,” ICLR2021

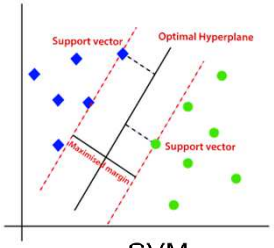
[출처] Open DMQA Seminar, Transformer in Computer Vision, Korea Univ.

- Apply a standard Transformer directly to images, with the fewest possible modifications.

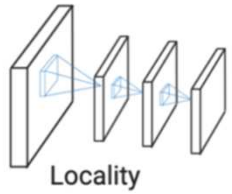
1) **Split an image into patches** and provide **the sequence of linear embeddings of these patches as an input** to a Transformer
2) **Treat image patches the same way as tokens (words)** in an NLP tasks. Train the model on image classification.

- When **trained on mid-sized datasets** (ImageNet) without strong regularization, ViT yield modest accuracies of a few percentage points below ResNets of comparable size.
- **Transformers *lack some of the inductive biases*** inherent to CNNs (such as *translation equivalence* and *locality*), and therefore do **not generalize well when trained on insufficient amounts of data.**
- **Large scale training trumps inductive bias. ViT attains excellent results when pre-trained at sufficient scale and transferred to tasks with fewer datapoints.** (Pretrained on **ImageNet-21K** or **JFT-300M** datasets)

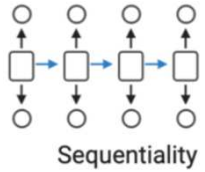
- ***Inductive Bias*** (or learning bias) : 학습 모델이 지금까지 만나보지 못했던 상황에서 정확한 예측을 하기 위해 사용하는 추가적인 가정을 의미
- SVM – Margin 최대화, CNN – 지역적 정보, RNN – 순차적 정보



SVM

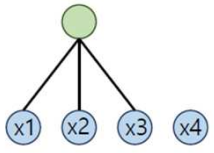


CNN

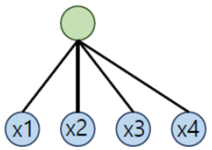


RNN

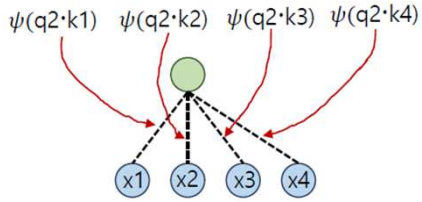
- Transformer : Inductive bias ↓, 모델의 자유도 ↑
  - ✓ 1차원 벡터로 만든 후 self attention (Global, 2차원의 지역적 정보 x)
  - ✓ Weight이 Input에 따라 유동적으로 변함
- CNN : 2차원의 지역적 특성 유지, 학습 후 Weight 고정



Convolution layer
- Locally connected
- Same weights for all inputs



Dense layer
- Fully connected
- Same weights for all inputs



Attention layer
- Fully connected
- Different weights for all inputs

1 ViT (Vision Transformer)

A. Dosovitskiy, et al. “An Image Is Worth 16X16 Words: Transformers for Image Recognition at Scale,” ICLR2021

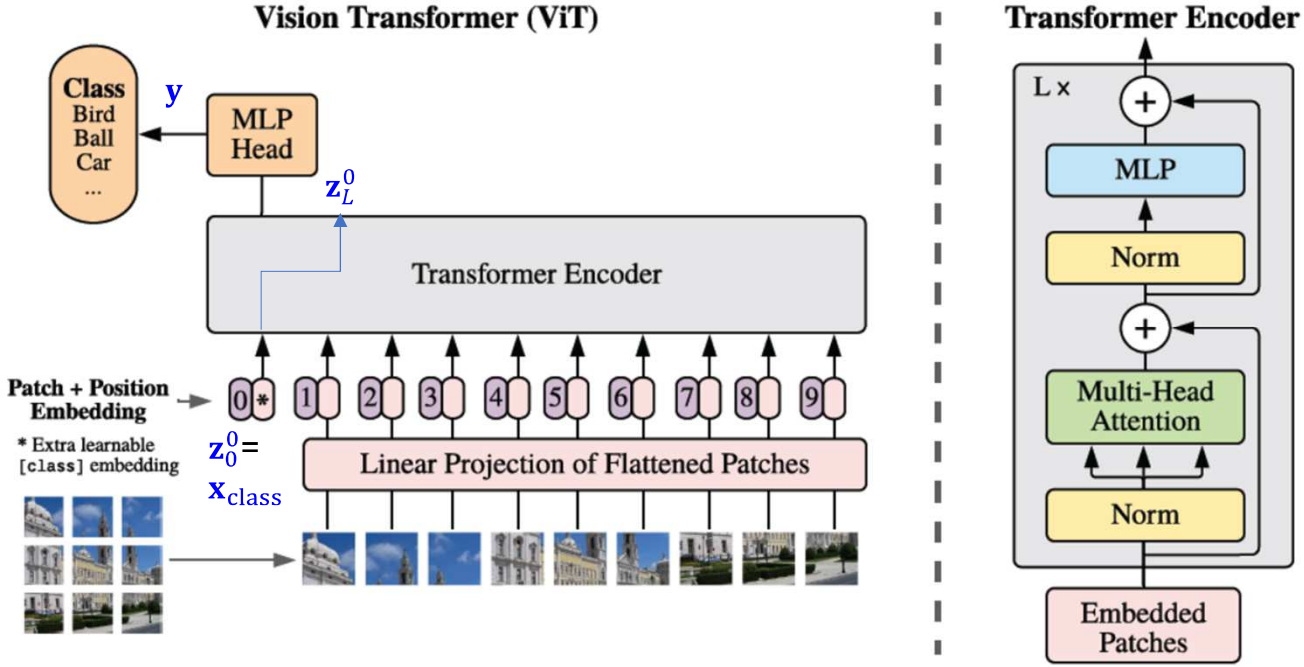


Figure 1: Model overview. We split an image into fixed-size patches, linearly embed each of them, add position embeddings, and feed the resulting sequence of vectors to a standard Transformer encoder. In order to perform classification, we use the standard approach of adding an extra learnable “*classification token*” to the sequence. The illustration of the Transformer encoder was inspired by Vaswani et al. (2017).

- Reshape an image $\mathbf{x} \in \mathbb{R}^{H \times W \times C}$ into a sequence of flattened 2D patches $\mathbf{x}_p \in \mathbb{R}^{N \times (P^2 \cdot C)}$

  (P, P) : Resolution of each patch

  $N = HW/P^2$: Number of patches (serve as the effective input sequence length for Transformer)

Embed size

- Transformer uses constant latent vector size D through all of its layers. → Flatten the patches and map to D dimensions with a trainable linear projection (*Patch embeddings*)

$$\mathbf{z}_0 = \left[\mathbf{x}_{\text{class}}; \mathbf{x}_p^1\mathbf{E}; \mathbf{x}_p^2\mathbf{E}; \cdots; \mathbf{x}_p^N\mathbf{E}\right] + \mathbf{E}_{\text{pos}} \quad (1)$$

$$\mathbf{E} \in \mathbb{R}^{(P^2 \cdot C) \times D}, \ \mathbf{E}_{pos} \in \mathbb{R}^{(N+1) \times D}$$

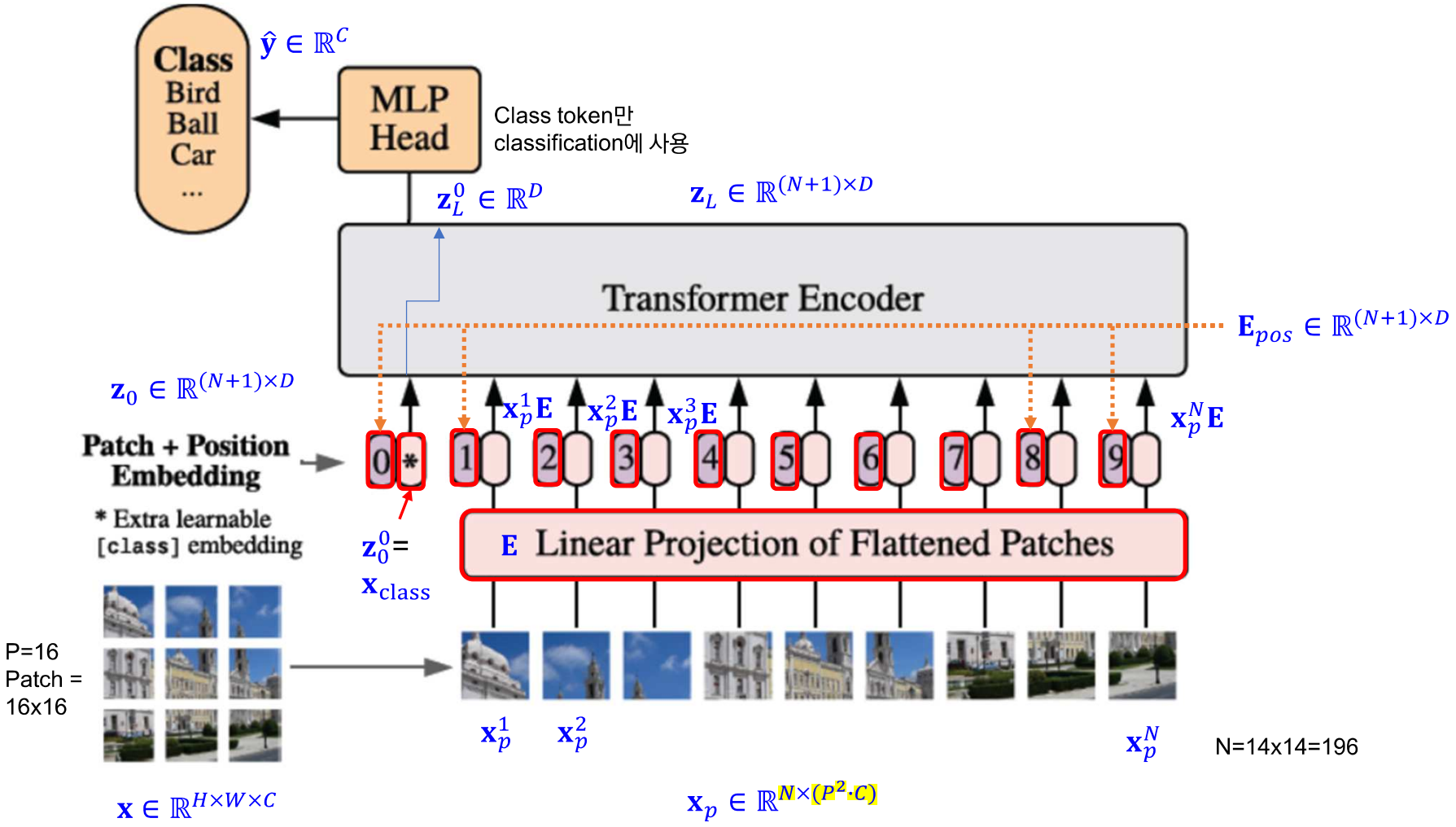
where $\mathbf{E}$ is the patch embedding projection

- Prepend a *learnable embedding* (Similar *class token*) to the sequence of embedded patches ($\mathbf{z}_0^0 = \mathbf{x}_{\text{class}}$), whose state at the output of Transformer Encoder ($\mathbf{z}_L^0$), serve as the image representation $\mathbf{y}$

- Both during pre-training and fine-tuning, a classification head is attached to $\mathbf{z}_L^0$. The classification head is implemented by a MLP with one hidden layer at pre-training time and by a single linear layer at fine-tuning time.

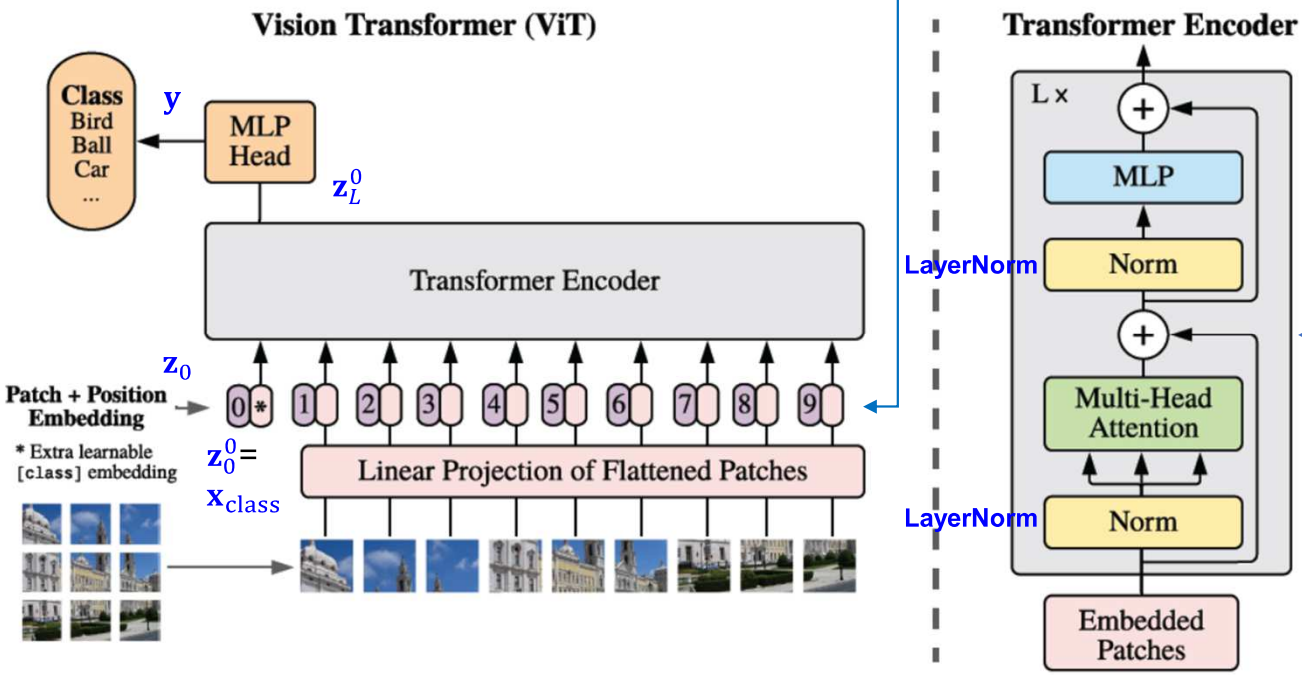
1 ViT (Vision Transformer)

A. Dosovitskiy, et al. “An Image Is Worth 16X16 Words: Transformers for Image Recognition at Scale,” ICLR2021



1 ViT (Vision Transformer)

A. Dosovitskiy, et al. "An Image Is Worth 16X16 Words: Transformers for Image Recognition at Scale," ICLR2021



$$\mathbf{z}_0 = [\mathbf{x}_{\text{class}}; \mathbf{x}_p^1\mathbf{E}; \mathbf{x}_p^2\mathbf{E}; \cdots; \mathbf{x}_p^N\mathbf{E}] + \mathbf{E}_{\text{pos}} \quad (1)$$

$$\mathbf{z}'_l = MSA(LN(\mathbf{z}_{l-1})) + \mathbf{z}_{l-1} \quad l = 1, \ldots, L \quad (2)$$

$$\mathbf{z}_l = MLP(LN(\mathbf{z}'_l)) + \mathbf{z}'_l \quad (3)$$

$$\mathbf{y} = LN(\mathbf{z}_L^0) \quad (4)$$

- MLP contains two layers with a *GELU non-linearity*.
- Both during pre-training and fine-tuning, a classification head is attached to $\mathbf{z}_L^0$.
- The classification head is implemented by a MLP with one hidden layer at pre-training time and by a single linear layer at fine-tuning time.

- Add Position Embedding to the patch embeddings; use standard learnable 1D position embeddings. → The resulting sequence of embedding vectors serves as input to Transformer Encoder.

- LayerNorm (LN) is applied before every block, and residual connections after every block

➢ GELU (Gaussian Error Linear Units)

$$\text{GELU}(x) = xP(X \le x) = x\Phi(x)$$
$$\approx 0.5x\left(1 + \tanh\left[\sqrt{2/\pi}\left(x + 0.044715x^3\right)\right]\right)$$

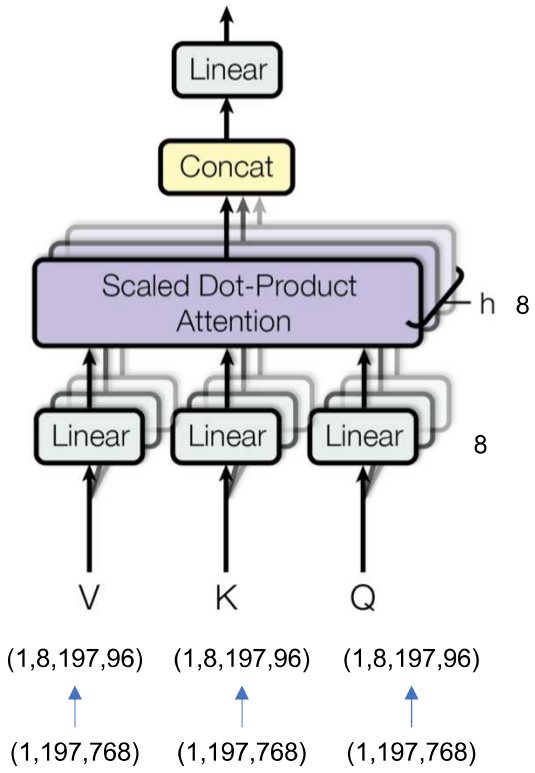
- LayerNorm : 각 Feature에 normalization (D dimension)

$$z_i = [z_i^1, z_i^2, z_i^3, \cdots, z_i^N, z_i^{N+1}] \qquad \text{LN}(z_i^j) = \gamma\frac{z_i^j - \mu_i}{\sigma_i} + \beta \quad = \gamma\frac{z_i^j - \mu_i}{\sqrt{\sigma_i^2 + \epsilon}} + \beta$$

1 ViT (Vision Transformer)

A. Dosovitskiy, et al. “An Image Is Worth 16X16 Words: Transformers for Image Recognition at Scale,” ICLR2021

MHA (Multi Head Attention)



MSA (Multi Head Self-Attention)

Reshape an image $\mathbf{x} \in \mathbb{R}^{H\times W\times C}$ into a sequence of flattened 2D patches $\mathbf{x}_p \in \mathbb{R}^{N\times(P^2\cdot C)}$

(P,P) : Resolution of each patch, $N = HW/P^2$: Number of patches

Appendix

Standard $\mathbf{qkv}$ self-attention (SA, Vaswani et al. (2017)) is a popular building block for neural architectures. For each element in an input sequence $\mathbf{z} \in \mathbb{R}^{N\times D}$, we compute a weighted sum over all values $\mathbf{v}$ in the sequence. The attention weights A_{ij} are based on the pairwise similarity between two elements of the sequence and their respective query $\mathbf{q}^i$ and key $\mathbf{k}^j$ representations.

$$[\mathbf{q},\mathbf{k},\mathbf{v}] = \mathbf{z}\mathbf{U}_{qkv} \qquad \mathbf{U}_{qkv} \in \mathbb{R}^{D\times 3D_h}, \tag{5}$$

$$A = \mathrm{softmax}\left(\mathbf{q}\mathbf{k}^\top/\sqrt{D_h}\right) \qquad A \in \mathbb{R}^{N\times N}, \tag{6}$$

$$\mathrm{SA}(\mathbf{z}) = A\mathbf{v}\,. \tag{7}$$

Multihead self-attention (MSA) is an extension of SA in which we run k self-attention operations, called “heads”, in parallel, and project their concatenated outputs. To keep compute and number of parameters constant when changing k, D_h (Eq. 5) is typically set to D/k.

$$\mathrm{MSA}(\mathbf{z}) = [\mathrm{SA}_1(z);\mathrm{SA}_2(z);\cdots;\mathrm{SA}_k(z)]\,\mathbf{U}_{msa} \qquad \mathbf{U}_{msa} \in \mathbb{R}^{k\cdot D_h\times D} \tag{8}$$

MSA 구할 때, 각 head의 q,k,v 에 대한 연산을 따로 하지 않고 한번에 처리

$$\text{head 1} : q_1 = z\cdot w_q^1, k_1 = z\cdot w_k^1, v_1 = z\cdot w_v^1$$

$$\text{head 2} : q_2 = z\cdot w_q^2, k_2 = z\cdot w_k^2, v_2 = z\cdot w_v^2$$

head 에서 weight만 달라지게 되므로

$$\text{Single Head} : q,k,v \in \mathbb{R}^{N\times D_h} \rightarrow \text{Multi Head} : q,k,v \in \mathbb{R}^{N\times k\times D_h}$$

1 ViT (Vision Transformer)

A. Dosovitskiy, et al. “An Image Is Worth 16X16 Words: Transformers for Image Recognition at Scale,” ICLR2021

Model Variants

Model	Layers	Hidden size D	MLP size	Heads	Params
ViT-Base	12	768	3072	12	86M
ViT-Large	24	1024	4096	16	307M
ViT-Huge	32	1280	5120	16	632M

Table 1: Details of Vision Transformer model variants.

- ViT-L/16 means the “Large” variant with 16x16 input patch size.
- Note that the Transformer’s sequence length is inversely proportional to the square of the patch size, thus models with smaller patch size are computationally more expensive.

Results

		ViT-B/16	ViT-B/32	ViT-L/16	ViT-L/32	ViT-H/14
ImageNet	CIFAR-10	98.13	97.77	97.86	97.94	-
	CIFAR-100	87.13	86.31	86.35	87.07	-
	ImageNet	77.91	73.38	76.53	71.16	-
	ImageNet ReaL	83.57	79.56	82.19	77.83	-
	Oxford Flowers-102	89.49	85.43	89.66	86.36	-
	Oxford-IIIT-Pets	93.81	92.04	93.64	91.35	-
ImageNet-21k	CIFAR-10	98.95	98.79	99.16	99.13	99.27
	CIFAR-100	91.67	91.97	93.44	93.04	93.82
	ImageNet	83.97	81.28	85.15	80.99	85.13
	ImageNet ReaL	88.35	86.63	88.40	85.65	88.70
	Oxford Flowers-102	99.38	99.11	99.61	99.19	99.51
	Oxford-IIIT-Pets	94.43	93.02	94.73	93.09	94.82
JFT-300M	CIFAR-10	99.00	98.61	99.38	99.19	99.50
	CIFAR-100	91.87	90.49	94.04	92.52	94.55
	ImageNet	84.15	80.73	87.12	84.37	88.04
	ImageNet ReaL	88.85	86.27	89.99	88.28	90.33
	Oxford Flowers-102	99.56	99.27	99.56	99.45	99.68
	Oxford-IIIT-Pets	95.80	93.40	97.11	95.83	97.56

Table 5: Top1 accuracy (in %) of Vision Transformer on various datasets when pre-trained on ImageNet, ImageNet-21k or JFT300M. These values correspond to Figure 3 in the main text. Models are fine-tuned at 384 resolution. Note that the ImageNet results are computed without additional techniques (Polyak averaging and 512 resolution images) used to achieve results in Table 2.

1 ViT (Vision Transformer)

A. Dosovitskiy, et al. “An Image Is Worth 16X16 Words: Transformers for Image Recognition at Scale,” ICLR2021

Results

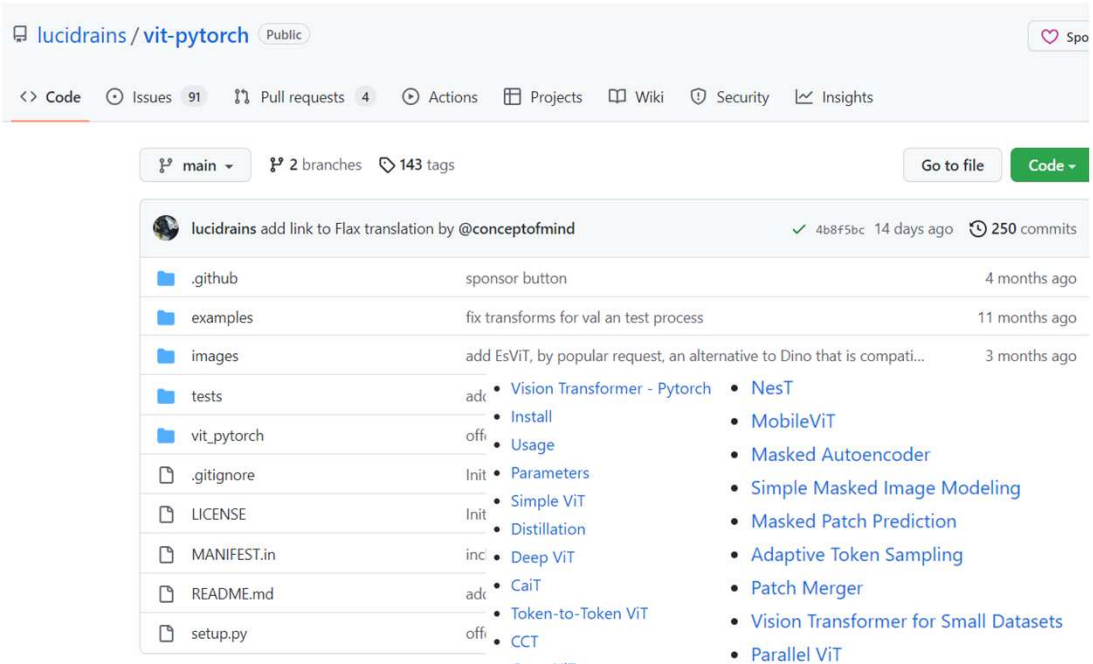
name	Epochs	ImageNet	ImageNet ReaL	CIFAR-10	CIFAR-100	Pets	Flowers	exaFLOPs
ViT-B/32	7	80.73	86.27	98.61	90.49	93.40	99.27	55
ViT-B/16	7	84.15	88.85	99.00	91.87	95.80	99.56	224
ViT-L/32	7	84.37	88.28	99.19	92.52	95.83	99.45	196
ViT-L/16	7	86.30	89.43	99.38	93.46	96.81	99.66	783
ViT-L/16	14	87.12	89.99	99.38	94.04	97.11	99.56	1567
ViT-H/14	14	88.08	90.36	99.50	94.71	97.11	99.71	4262
ResNet50x1	7	77.54	84.56	97.67	86.07	91.11	94.26	50
ResNet50x2	7	82.12	87.94	98.29	89.20	93.43	97.02	199
ResNet101x1	7	80.67	87.07	98.48	89.17	94.08	95.95	96
ResNet152x1	7	81.88	87.96	98.82	90.22	94.17	96.94	141
ResNet152x2	7	84.97	89.69	99.06	92.05	95.37	98.62	563
ResNet152x2	14	85.56	89.89	99.24	91.92	95.75	98.75	1126
ResNet200x3	14	87.22	90.15	99.34	93.53	96.32	99.04	3306
R50x1+ViT-B/32	7	84.90	89.15	99.01	92.24	95.75	99.46	106
R50x1+ViT-B/16	7	85.58	89.65	99.14	92.63	96.65	99.40	274
R50x1+ViT-L/32	7	85.68	89.04	99.24	92.93	96.97	99.43	246
R50x1+ViT-L/16	7	86.60	89.72	99.18	93.64	97.03	99.40	859
R50x1+ViT-L/16	14	87.12	89.76	99.31	93.89	97.36	99.11	1668

Table 6: Detailed results of model scaling experiments. These correspond to Figure 5 in the main paper. We show transfer accuracy on several datasets, as well as the pre-training compute (in exaFLOPs).

1 ViT (Visual Transformer) in PyTorch

lucidrains / vit-pytorch

https://github.com/lucidrains/vit-pytorch



- Vision Transformer - Pytorch
- Install
- Usage
- Parameters
- Simple ViT
- Distillation
- Deep ViT
- CaiT
- Token-to-Token ViT
- CCT
- Cross ViT
- PiT
- LeViT
- CvT
- Twins SVT
- CrossFormer
- RegionViT
- ScalableViT
- SepViT
- MaxViT
- NesT
- MobileViT
- Masked Autoencoder
- Simple Masked Image Modeling
- Masked Patch Prediction
- Adaptive Token Sampling
- Patch Merger
- Vision Transformer for Small Datasets
- Parallel ViT
- Learnable Memory ViT
- Dino
- EsViT
- Accessing Attention
- Research Ideas
  - Efficient Attention
  - Combining with other Transformer improvements
- FAQ
- Resources
- Citations

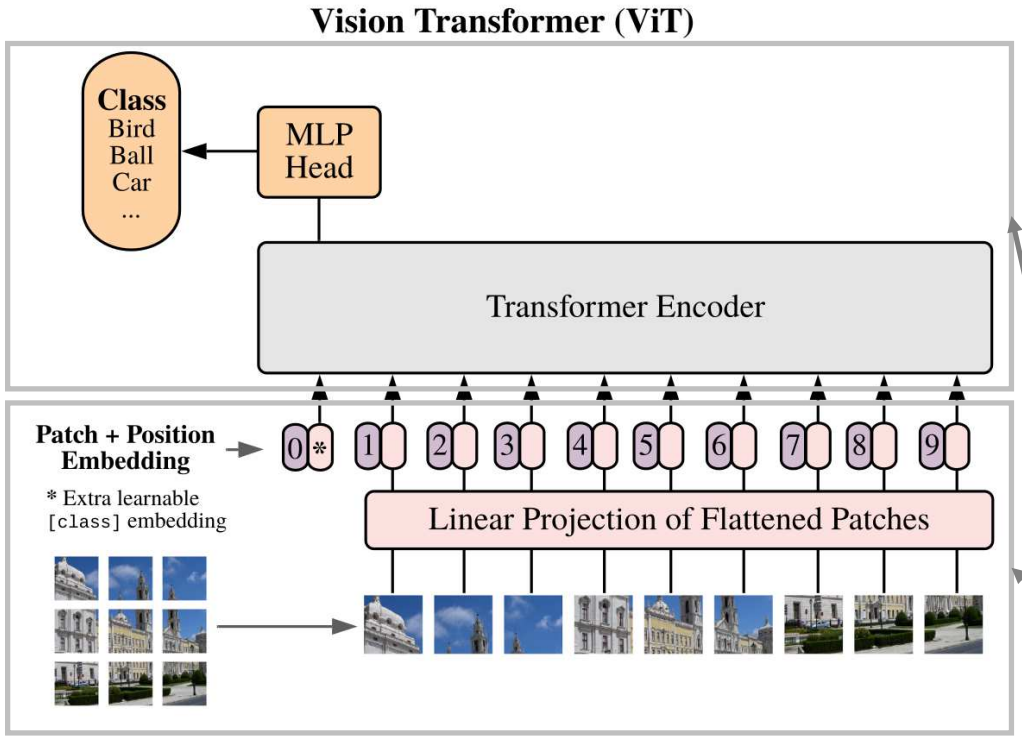
1 ViT (Visual Transformer) in PyTorch

[Google Research, Brain Team]

A. Dosovitskiy, et al. “An Image Is Worth 16X16 Words: Transformers for Image Recognition at Scale,” ICLR2021

[출처] https://towardsdatascience.com/implementing-visualttransformer-in-pytorch-184f9f16f632
[Code] https://github.com/FrancescoSaverioZuppichini/ViT

- ViT uses a normal transformer (the one proposed in Attention is All You Need) that works on images.



Organization Sections

- Data
- Patches Embeddings
  - CLS Token
  - Position Embedding
- Transformer
  - Attention
  - Residuals
  - MLP
  - TransformerEncoder
- Head
- ViT

4) The resulting tensor is passed first into a standard Transformer and then to a classification head.

1) Decompose the input image into 16x16 flatten patches (the image is not in scale).
2) Then embed them using a normal fully connected layer,
3) Add a special cls token in front of them and sum the positional encoding.

1 ViT (Visual Transformer) in PyTorch

[출처] https://towardsdatascience.com/implementing-visualttransformer-in-pytorch-184f9f16f632
[Code] https://github.com/FrancescoSaverioZuppichini/ViT

Prerequisite

- Have a look at the amazing The Illustrated Transformer website
- Watch Yannic Kilcher video about ViT
- Read Einops doc

```
pip install eniops
```

Packages

```
import torch
import torch.nn.functional as F
import matplotlib.pyplot as plt

from torch import nn
from torch import Tensor
from PIL import Image
from torchvision.transforms import Compose, Resize, ToTensor
from einops import rearrange, reduce, repeat
from einops.layers.torch import Rearrange, Reduce
from torchsummary import summary
```

Sample Data

- ViT uses a normal transformer (the one proposed in Attention is All You Need) that works on images.

```
img = Image.open('./cat.jpg')
fig = plt.figure()
plt.imshow(img)

# resize to imagenet size
transform = Compose([Resize((224, 224)), ToTensor()])
x = transform(img)
x = x.unsqueeze(0) # add batch dim
```

```
x.shape : torch.Size([1, 3, 224, 224])
```

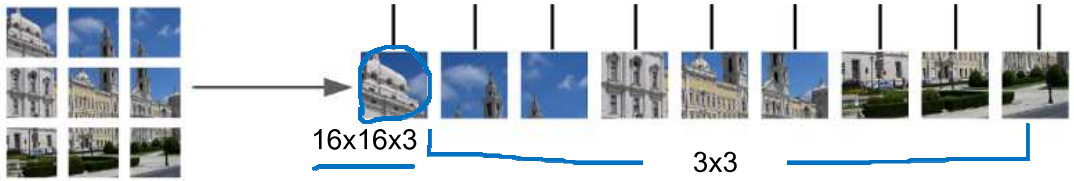
1 ViT (Visual Transformer) in PyTorch

[출처] https://towardsdatascience.com/implementing-visualttransformer-in-pytorch-184f9f16f632
[Code] https://github.com/FrancescoSaverioZuppichini/ViT

1. Patch Embeddings

1) Patches

- The first step is to break-down the image in multiple patches and flatten them.



b c (h s1) (w s2) = 1, 3, (3x16), (3x16) b c (h w) (s1 s2 c) = 1, (3x3), (16x16x3)

- Reshape

$$\mathbf{x} \in \mathbb{R}^{H \times W \times C} \longrightarrow \mathbf{x}_p \in \mathbb{R}^{N \times (P^2 \cdot C)}$$

2D image A sequence of flattened 2D patches

$P \times P$: resolution of each patch (16, 16)

$N = HW/P^2$ Number of patches

- This can be easily done using einops.

```
patch_size = 16 # 16 pixels
patches = rearrange(x, 'b c (h s1) (w s2) -> b (h w) (s1 s2 c)', s1=patch_size, s2=patch_size)
```

```
x.shape : torch.Size([1, 3, 224, 224])
```

```
patches.shape : torch.Size([1, 196, 768])
```

b c (h s1) (w s2) = 1, 3, (14x16), (14x16)
b (h w) (s1 s2 c) = 1, (14x14), (16x16x3)

An overview of the model is depicted in Figure 1. The standard Transformer receives as input a 1D sequence of token embeddings. To handle 2D images, we reshape the image $\mathbf{x} \in \mathbb{R}^{H \times W \times C}$ into a sequence of flattened 2D patches $\mathbf{x}_p \in \mathbb{R}^{N \times (P^2 \cdot C)}$, where (H, W) is the resolution of the original image, C is the number of channels, (P, P) is the resolution of each image patch, and $N = HW/P^2$ is the resulting number of patches, which also serves as the effective input sequence length for the Transformer. The Transformer uses constant latent vector size D through all of its layers, so we

einops

[주 출처] https://github.com/arogozhnikov/einops/
[참고] https://yhkim4504.tistory.com/5
[중요] https://rockt.github.io/2018/04/30/einsum

einops

[중요 참고] EINSUM IS ALL YOU NEED - EINSTEIN SUMMATION IN DEEP LEARNING
https://rockt.github.io/2018/04/30/einsum

Tutorials

- Part 1: einops fundamentals
- Part 2: einops for deep learning
- Part 3: real code fragments improved with einops (so far only for pytorch)

```
from einops import rearrange, reduce, repeat
# rearrange elements according to the pattern
output_tensor = rearrange(input_tensor, 't b c -> b c t')
# combine rearrangement and reduction
output_tensor = reduce(input_tensor, 'b c (h h2) (w w2) -> b h w c', 'mean', h2=2, w2=2)
# copy along a new axis
output_tensor = repeat(input_tensor, 'h w -> h w c', c=3)
```

1 ViT (Visual Transformer) in PyTorch

[출처] https://towardsdatascience.com/implementing-visualttransformer-in-pytorch-184f9f16f632
[Code] https://github.com/FrancescoSaverioZuppichini/ViT

1) Patches

- Create a PatchEmbedding class

Ver1

```
class PatchEmbedding(nn.Module):
    def __init__(self, in_channels: int = 3, patch_size: int = 16, emb_size: int = 768):
        self.patch_size = patch_size
        super().__init__()
        self.projection = nn.Sequential(
            # break-down the image in s1 x s2 patches and flat them
            Rearrange('b c (h s1) (w s2) -> b (h w) (s1 s2 c)', s1=patch_size,
s2=patch_size),
            nn.Linear(patch_size * patch_size * in_channels, emb_size)
        )

    def forward(self, x: Tensor) -> Tensor:
        x = self.projection(x)
        return x

PatchEmbedding()(x).shape
```

```
torch.nn.Conv2d(in_channels, out_channels, kernel_size, stride=1,
padding=0, dilation=1, groups=1, bias=True, padding_mode='zeros')
```

Ver2

```
class PatchEmbedding(nn.Module):
    def __init__(self, in_channels: int = 3, patch_size: int = 16, emb_size: int = 768):
        self.patch_size = patch_size
        super().__init__()
        self.projection = nn.Sequential(
            # using a conv layer instead of a linear one -> performance gains
            nn.Conv2d(in_channels, emb_size, kernel_size=patch_size,
stride=patch_size),
            Rearrange('b e (h) (w) -> b (h w) e'),
        )

    def forward(self, x: Tensor) -> Tensor:
        x = self.projection(x)
        return x

PatchEmbedding()(x).shape
```

- [Note] After checking out the original implementation, I found out that the authors are using a Conv2d layer instead of a Linear one for performance gain. This is obtained by using a kernel_size and stride equal to the `patch_size`. Intuitively, the convolution operation is applied to each patch individually. So, we have to first apply the conv layer and then flat the resulting images.

1 ViT (Visual Transformer) in PyTorch

[출처] https://towardsdatascience.com/implementing-visualttransformer-in-pytorch-184f9f16f632
[Code] https://github.com/FrancescoSaverioZuppichini/ViT

1) Patches

- Create a PatchEmbedding class

```
torch.nn.Conv2d(in_channels, out_channels, kernel_size, stride=1, padding=0, dilation=1, groups=1, bias=True, padding_mode='zeros')
```

```
patches = rearrange(x, 'b c (h s1) (w s2) -> b (h w) (s1 s2 c)', s1=patch_size, s2=patch_size)
patches.shape
```

```
torch.Size([1, 196, 768])
```

```
linear = nn.Linear(patch_size * patch_size * 3, 768)
linear(patches).shape
```

```
torch.Size([1, 196, 768])
```

```
conv1 = nn.Conv2d(3, 768, 16, 16)
conv1(x).shape
```

```
torch.Size([1, 768, 14, 14])
```

```
rearrang = Rearrange('b e (h) (w) -> b (h w) e')
rearrang(conv1(x)).shape
```

```
torch.Size([1, 196, 768])
```

b c (h s1) (w s2) = [1, 3, (14x16), (14x16)] = [1, 3, 224, 224]

b c (h w) (s1 s2 c) = 1, (14x14), (16x16x3) = [1, 196, 768]

conv1 = [1, 768, 14, 14]

b e (h) (w) = [1, 768, 14, 14]

b (h w) e = [1, 14x14, 768] = [1, 196, 768]

1 ViT (Visual Transformer) in PyTorch

[출처] https://towardsdatascience.com/implementing-visualttransformer-in-pytorch-184f9f16f632
[Code] https://github.com/FrancescoSaverioZuppichini/ViT

2) CLS Token

- Next, add the cls token and the position embedding.
- The cls token is just a number placed in from of **each** sequence (of projected patches)

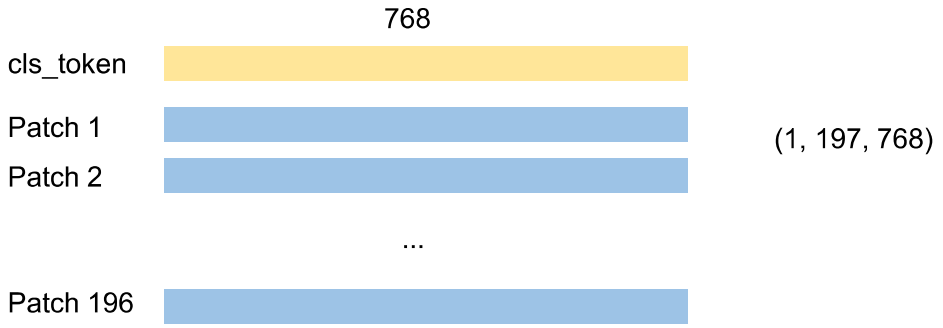
Ver3

```
class PatchEmbedding(nn.Module):
    def __init__(self, in_channels: int = 3, patch_size: int = 16, emb_size: int = 768):
        self.patch_size = patch_size
        super().__init__()
        self.proj = nn.Sequential(
            # using a conv layer instead of a linear one -> performance gains
            nn.Conv2d(in_channels, emb_size, kernel_size=patch_size,
stride=patch_size),
            Rearrange('b e (h) (w) -> b (h w) e'),
        )

        self.cls_token = nn.Parameter(torch.randn(1,1, emb_size))

    def forward(self, x: Tensor) -> Tensor:
        b, _, _, _ = x.shape
        x = self.proj(x)
        cls_tokens = repeat(self.cls_token, '() n e -> b n e', b=b)
        # prepend the cls token to the input
        x = torch.cat([cls_tokens, x], dim=1)
        return x

PatchEmbedding()(x).shape
```

```
torch.Size([1, 197, 768])
```

```
emb_size = 768
b=1
cls_token = nn.Parameter(torch.randn(1,1, emb_size))
cls_token.shape
cls_tokens = repeat(cls_token, '() n e -> b n e', b=b)
cls_tokens.shape
```

```
torch.Size([1, 1, 768])

torch.Size([1, 1, 768])
```



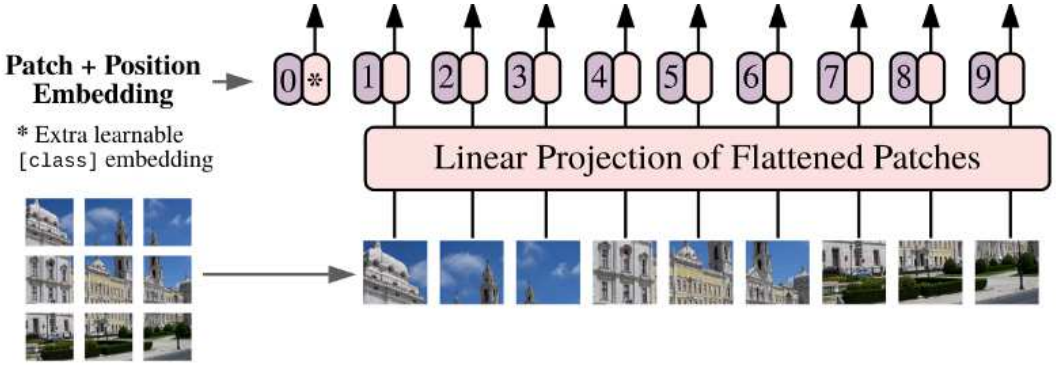
- cls_token is a torch Parameter randomly initialized, in the forward the method it is copied b (batch) times and prepended before the projected patches using torch.cat

1 ViT (Visual Transformer) in PyTorch

[출처] https://towardsdatascience.com/implementing-visualttransformer-in-pytorch-184f9f16f632
[Code] https://github.com/FrancescoSaverioZuppichini/ViT

3) Position Embedding

- So far, the model has no idea about the original position of the patches. We need to pass this spatial information. This can be done in different ways, in ViT we let the model learn it. The position embedding is just a tensor of shape (N_PATCHES + 1 (token), EMBED_SIZE) that is added to the projected patches.



- We added the position embedding in the .positions field and sum it to the patches in the .forward function

```
emb_size = 768
img_size = 224
patch_size = 16
b=1
```

```
positions = nn.Parameter(torch.randn((img_size // patch_size) **2 + 1, emb_size))
positions.shape
```

```
torch.Size([197, 768])
```

Ver4

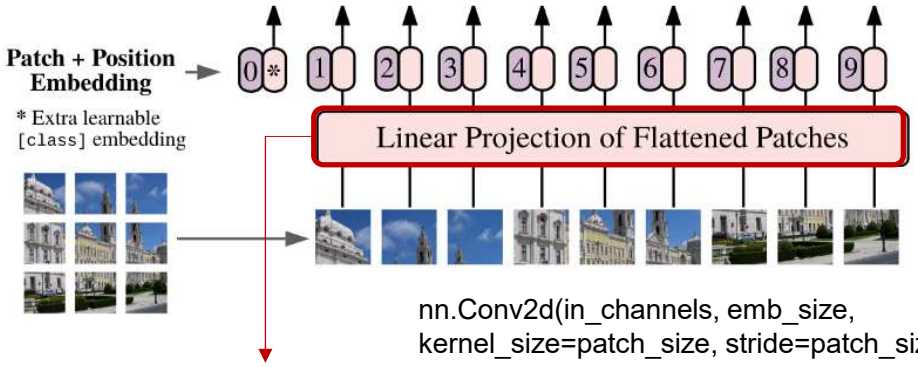
```
class PatchEmbedding(nn.Module):
    def __init__(self, in_channels: int = 3, patch_size: int = 16, emb_size: int = 768, img_size: int = 224):
        self.patch_size = patch_size
        super().__init__()
        self.projection = nn.Sequential(
            # using a conv layer instead of a linear one -> performance gains
            nn.Conv2d(in_channels, emb_size, kernel_size=patch_size, stride=patch_size),
            Rearrange('b e (h) (w) -> b (h w) e'),
        )
        self.cls_token = nn.Parameter(torch.randn(1,1, emb_size))
        self.positions = nn.Parameter(torch.randn((img_size // patch_size) **2 + 1, emb_size))
                                                                    (224//16)^2+1=14^2+1=197
            768

    def forward(self, x: Tensor) -> Tensor:
        b, _, _, _ = x.shape
        x = self.projection(x)
        cls_tokens = repeat(self.cls_token, '() n e -> b n e', b=b)
        # prepend the cls token to the input
        x = torch.cat([cls_tokens, x], dim=1)
        # add position embedding
        x += self.positions
        return x

PatchEmbedding()(x).shape
```

```
torch.Size([1, 197, 768])
```

1 ViT (Visual Transformer) in PyTorch

[출처] https://towardsdatascience.com/implementing-visualttransformer-in-pytorch-184f9f16f632
[Code] https://github.com/FrancescoSaverioZuppichini/ViT

Inspecting Vision Transformer



nn.Conv2d(in_channels, emb_size, kernel_size=patch_size, stride=patch_size),

self.positions = nn.Parameter(torch.randn((img_size // patch_size) **2 + 1, emb_size))

RGB embedding filters (first 28 principal components)

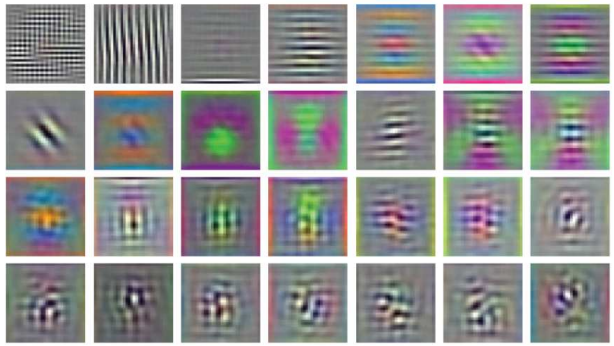


Fig. 7. Filters of the initial linear embedding of RGB values of ViT-L/32.

The first layer of ViT linearly projects the flattened patches into a lower-dimensional space.

Top principal components of the learned embedding filters. – Resemble plausible basis function for a low-dimensional representation of the fine structure within each patch.

$$\mathbf{z}_0 = \left[\mathbf{x}_{\text{class}}; \mathbf{x}_p^1\mathbf{E}; \mathbf{x}_p^2\mathbf{E}; \cdots; \mathbf{x}_p^N\mathbf{E}\right] + \mathbf{E}_{\text{pos}}$$

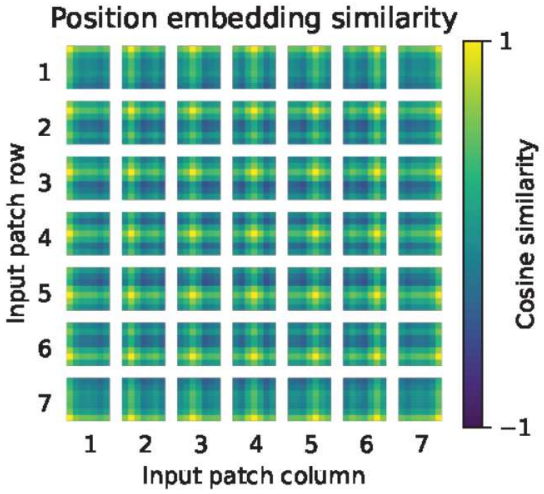


Fig. 7. Similarity of position embeddings of ViT-L/32.

Tiles show the cosine similarity between the position embedding of the patch with the indicated row and column and the position embeddings of all other patches.

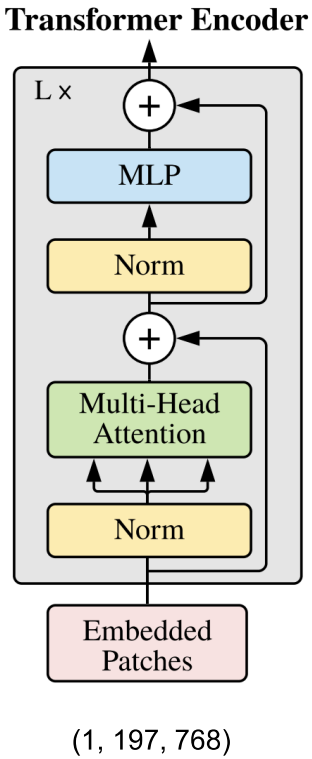
The model learns to encode distance between the image in the similarity of position embeddings. Closer patches tend to have more similar position embeddings.

1 ViT (Visual Transformer) in PyTorch

[출처] https://towardsdatascience.com/implementing-visualttransformer-in-pytorch-184f9f16f632
[출처] https://yhkim4504.tistory.com/6
[출처] https://heung-bae-lee.github.io/2020/01/21/deep_learning_10/

2. Transformer Encoder

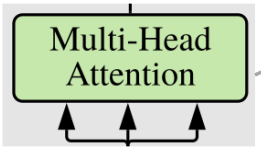
- Now we need the implement Transformer. In ViT only the Encoder is used, the architecture is visualized in the following picture.



(1, 197, 768)

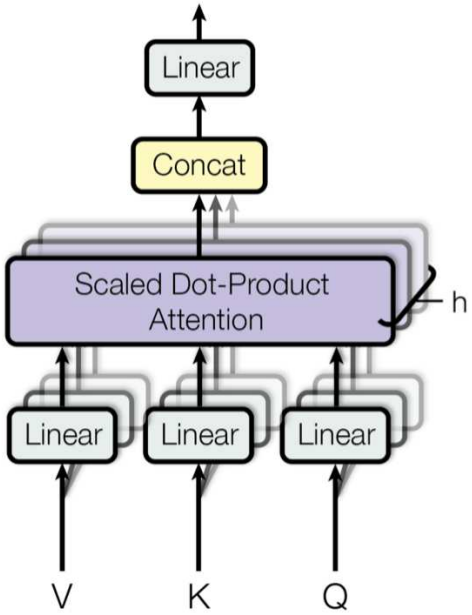
1) Multi Head Attention

- The attention takes three inputs, the famous *queries*, *keys*, and *values*, and computes the attention matrix using queries and values and use it to "attend" to the values. In this case, we are using multi-head attention meaning that the computation is split across n heads with smaller input size.



- 4 fully connected layers, one for queries, keys, values, and a final one dropout.

MHA (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

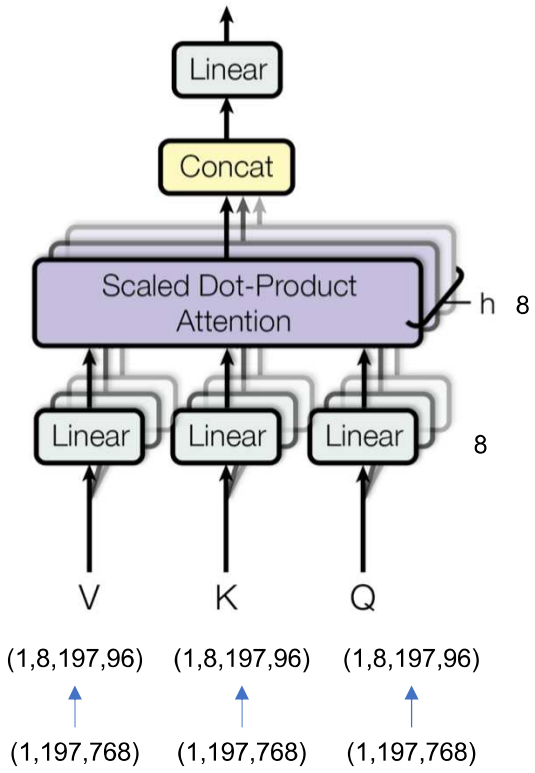
$$\text{Linear}_i(V) = VW_{V,i} \quad W_{V,i} \in \mathbb{R}^{d_v \times d_{model}}$$
$$\text{Linear}_i(K) = KW_{K,i} \quad W_{K,i} \in \mathbb{R}^{d_k \times d_{model}}$$
$$\text{Linear}_i(Q) = QW_{Q,i} \quad W_{Q,i} \in \mathbb{R}^{d_q \times d_{model}}$$

ViT에서의 MHA는 QKV가 같은 텐서로 입력됨. 입력텐서는 3개의 Linear Projection을 통해 임베딩된 후 여러 개의 Head로 나눠진 후 각각 Scaled Dot-Product Attention을 진행함

1 ViT (Visual Transformer) in PyTorch

[출처] https://yhkim4504.tistory.com/6

MHA (Multi Head Attention)



(1) Linear Projection

```
emb_size = 768
num_heads = 8

keys = nn.Linear(emb_size, emb_size)
queries = nn.Linear(emb_size, emb_size)
values = nn.Linear(emb_size, emb_size)
print(keys, queries, values)
```

```
Linear(in_features=768, out_features=768, bias=True)
Linear(in_features=768, out_features=768, bias=True)
Linear(in_features=768, out_features=768, bias=True)
```

먼저 임베딩된 입력텐서를 받아서 다시 임베딩 사이즈로 Linear Projection을 하는 레이어를 3개 만듬. 입력 텐서를 QKV로 만드는 각 레이어는 모델 훈련과정에서 학습됨.

$$q = z \cdot w_q \quad (w_q \in \mathbb{R}^{D \times D_h})$$

$$k = z \cdot w_k \quad (w_k \in \mathbb{R}^{D \times D_h})$$

$$v = z \cdot w_v \quad (w_v \in \mathbb{R}^{D \times D_h})$$

$$[q, k, v] = z \cdot U_{qkv} \quad (U_{qkv} \in \mathbb{R}^{D \times 3D_h})$$

(2) Multi Head

```
z=PatchEmbedding()(x)

queries = rearrange(queries(z), "b n (h d) -> b h n d", h=num_heads)
keys = rearrange(keys(z), "b n (h d) -> b h n d", h=num_heads)
values  = rearrange(values(z), "b n (h d) -> b h n d", h=num_heads)

print('shape :', queries.shape, keys.shape, values.shape)
```

```
shape : torch.Size([1, 8, 197, 96]) torch.Size([1, 8, 197, 96])
        torch.Size([1, 8, 197, 96])
```

```
z=PatchEmbedding()(x)
queries(z).shape
```

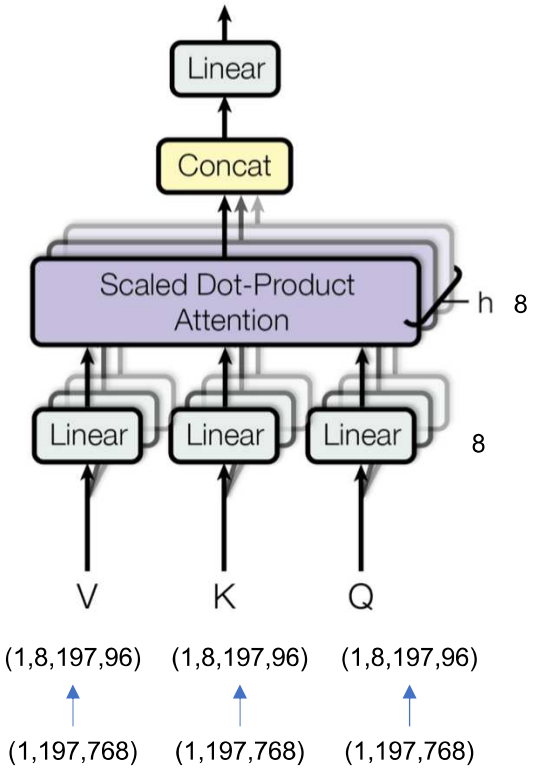
```
torch.Size([1, 197, 768])
```

이후 각 Linear Projection을 거친 QKV를 rearrange를 통해 8개의 Multi-Head로 나눠주게 됩니다.

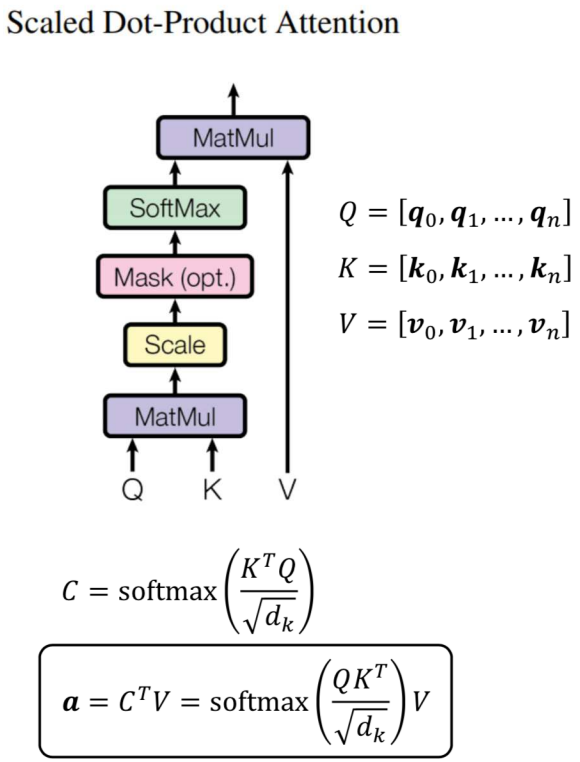
1 ViT (Visual Transformer) in PyTorch

[출처] https://yhkim4504.tistory.com/6
[출처] https://heung-bae-lee.github.io/2020/01/21/deep_learning_10/

MHA (Multi Head Attention)



(3) Scaled Dot Product Attention



위 그림과 같이 Q와 K를 곱합니다. einops를 이용해 자동으로 transpose 후 내적이 진행됩니다. 그 다음 scaling 해준 후 얻어진 Attention Score와 V를 내적하고 다시 emb_size로 rearrange 하면 MHA의 output이 나오게 됩니다.

```
# Queries * Keys
energy = torch.einsum('bhqd, bhkd -> bhqk', queries, keys)
print('energy :', energy.shape)

# Get Attention Score
scaling = emb_size ** (1/2)
att = F.softmax(energy, dim=-1) / scaling
print('att :', att.shape)

# Attention Score * values
out = torch.einsum('bhal, bhlv -> bhav ', att, values)
print('out :', out.shape)

# Rearrage to emb_size
out = rearrange(out, "b h n d -> b n (h d)")
print('out2 : ', out.shape)
```

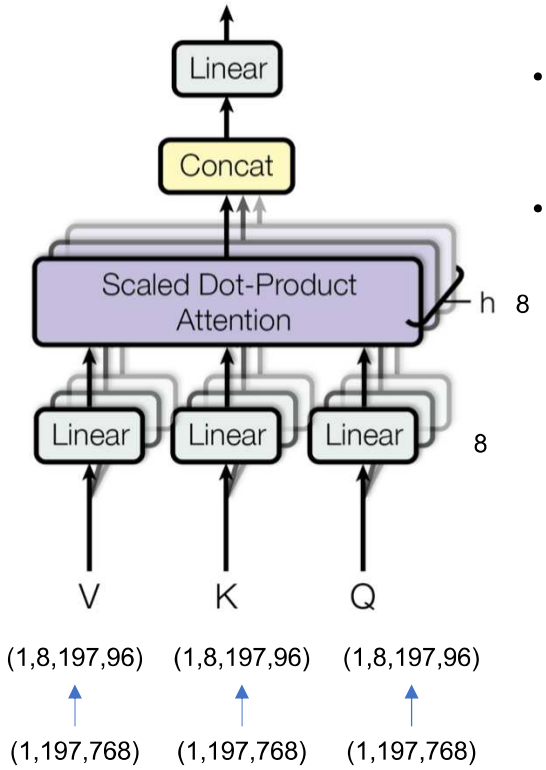
```
energy : torch.Size([1, 8, 197, 197])
att : torch.Size([1, 8, 197, 197])
out : torch.Size([1, 8, 197, 96])
out2 :  torch.Size([1, 197, 768])
```

torch.einsum
https://rockt.github.io/2018/04/30/einsum
https://baekyeongmin.github.io/dev/einsum/

1 ViT (Visual Transformer) in PyTorch

[출처] https://towardsdatascience.com/implementing-visualttransformer-in-pytorch-184f9f16f632
[Code] https://github.com/FrancescoSaverioZuppichini/ViT

2. Transformer Encoder



- 4 fully connected layers, one for queries, keys, values, and a final one dropout.
- The *forward* method takes as input the queries, keys, and values from the previous layer and projects them using the three linear layers. Since we implementing multi heads attention, we have to *rearrange* the result in multiple heads.

```
class MultiHeadAttention(nn.Module):
    def __init__(self, emb_size: int = 768, num_heads: int = 8, dropout: float = 0):
        super().__init__()
        self.emb_size = emb_size
        self.num_heads = num_heads
        self.keys = nn.Linear(emb_size, emb_size)
        self.queries = nn.Linear(emb_size, emb_size)
        self.values = nn.Linear(emb_size, emb_size)
        self.att_drop = nn.Dropout(dropout)
        self.projection = nn.Linear(emb_size, emb_size)

    def forward(self, x : Tensor, mask: Tensor = None) -> Tensor:
        # split keys, queries and values in num_heads
        queries = rearrange(self.queries(x), "b n (h d) -> b h n d", h=self.num_heads)
        keys = rearrange(self.keys(x), "b n (h d) -> b h n d", h=self.num_heads)
        values  = rearrange(self.values(x), "b n (h d) -> b h n d", h=self.num_heads)

        # sum up over the last axis
        # b : batch, h : num_heads, q : query_len, k : key_len
        energy = torch.einsum('bhqd, bhkd -> bhqk', queries, keys)
        if mask is not None:
            fill_value = torch.finfo(torch.float32).min  # -3.4028234663852886e+38
            energy.mask_fill(~mask, fill_value)

        scaling = self.emb_size ** (1/2)
        att = F.softmax(energy, dim=-1) / scaling
        att = self.att_drop(att)
        # sum up over the third axis
        out = torch.einsum('bhal, bhlv -> bhav ', att, values)
        out = rearrange(out, "b h n d -> b n (h d)")
        out = self.projection(out)
        return out
```

1 ViT (Visual Transformer) in PyTorch

[출처] https://towardsdatascience.com/implementing-visualttransformer-in-pytorch-184f9f16f632
[Code] https://github.com/FrancescoSaverioZuppichini/ViT

2. Transformer Encoder

Figure 6: Representative examples of attention from the output token to the input space. See Appendix D.7 for details.

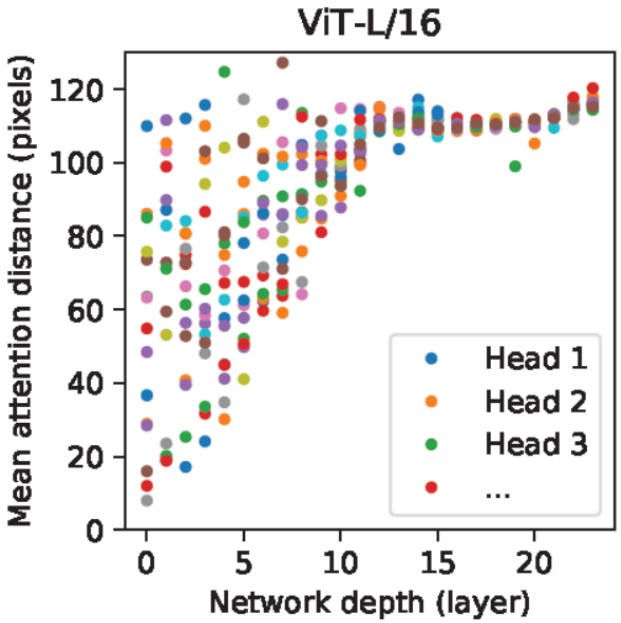


Fig. 7. Size of attended area by head and network depth. Each dot shows the mean attention distance across images for one of 16 heads at one layer.

Self-attention allows ViT to integrate information across the entire image even in the lowest layers.

We investigate to what degree the network makes use of this capability. Specifically, we compute the **average distance** in image space across which information is integrated, based on the attention weights (Figure 7, right). This "**attention distance**" is analogous to receptive field size in CNNs.

We find that some heads attend to most of the image already in the lowest layers, showing that the ability to integrate information globally is indeed used by the model.

Other attention heads have consistently small attention distances in the low layers. This highly localized attention is less pronounced in hybrid models that apply a ResNet before the Transformer (Figure 7, right), suggesting that it may serve a similar function as early convolutional layers in CNNs.

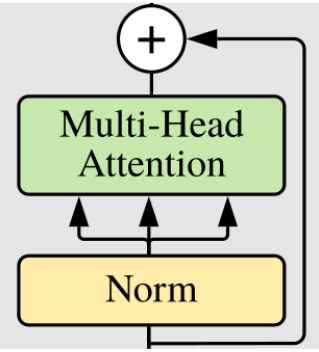
Further, the attention distance increases with network depth. Globally, we find that the model attends to image regions that are semantically relevant for classification (Figure 6).

1 ViT (Visual Transformer) in PyTorch

[출처] https://towardsdatascience.com/implementing-visualttransformer-in-pytorch-184f9f16f632
[Code] https://github.com/FrancescoSaverioZuppichini/ViT

2. Transformer Encoder

2) Residual



- The transformer block has residuals connection
- We can create a nice wrapper to perform the residual addition, it will be handy later on.

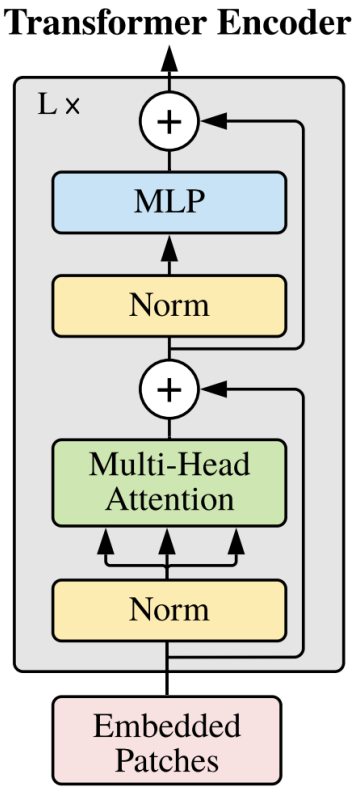
```
class ResidualAdd(nn.Module):
    def __init__(self, fn):
        super().__init__()
        self.fn = fn

    def forward(self, x, **kwargs):
        res = x
        x = self.fn(x, **kwargs)
        x += res
        return x
```

1 ViT (Visual Transformer) in PyTorch

[출처] https://towardsdatascience.com/implementing-visualttransformer-in-pytorch-184f9f16f632
[Code] https://github.com/FrancescoSaverioZuppichini/ViT

2. Transformer Encoder

3) MLP



- The attention's output is passed to a fully connected layer composed of two layers that upsample by a factor of *expansion* the input

- *Just a quick side note*. I don't know why but I've never seen people subclassing nn.Sequential to avoid writing the forward method. Start doing it, this is how object programming works!

- Finally, we can create the Transformer Encoder Block
- *ResidualAdd* allows us to define this block in an elegant way

```
class FeedForwardBlock(nn.Sequential):
    def __init__(self, emb_size: int, expansion: int = 4, drop_p: float = 0.):
        super().__init__(
            nn.Linear(emb_size, expansion * emb_size),
            nn.GELU(),
            nn.Dropout(drop_p),
            nn.Linear(expansion * emb_size, emb_size),
        )
```

```
class TransformerEncoderBlock(nn.Sequential):
    def __init__(self,
                 emb_size: int = 768,
                 drop_p: float = 0.,
                 forward_expansion: int = 4,
                 forward_drop_p: float = 0.,
                 ** kwargs):
        super().__init__(
            ResidualAdd(nn.Sequential(
                nn.LayerNorm(emb_size),
                MultiHeadAttention(emb_size, **kwargs),
                nn.Dropout(drop_p)
            )),
            ResidualAdd(nn.Sequential(
                nn.LayerNorm(emb_size),
                FeedForwardBlock(
                    emb_size, expansion=forward_expansion, drop_p=forward_drop_p),
                nn.Dropout(drop_p)
            )
            ))
```

1 ViT (Visual Transformer) in PyTorch

[출처] https://towardsdatascience.com/implementing-visualttransformer-in-pytorch-184f9f16f632
[Code] https://github.com/FrancescoSaverioZuppichini/ViT

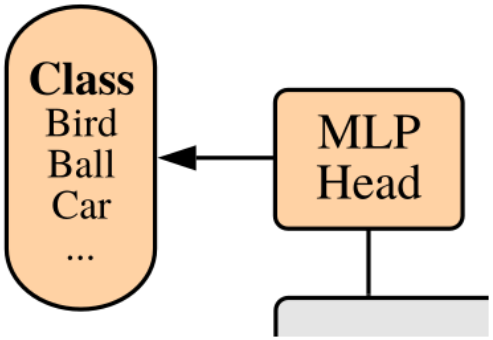
2. Transformer Encoder

- Test it.

```
patches_embedded = PatchEmbedding()(x)
TransformerEncoderBlock()(patches_embedded).shape
```

```
torch.Size([1, 197, 768])
```

- you can also *PyTorch build-in multi-head attention* but it will expect 3 inputs: queries, keys, and values. You can subclass it and pass the same input



Transformer

- In ViT only the Encoder part of the original transformer is used. Easily, the encoder is L blocks of TransformerBlock.

```
class TransformerEncoder(nn.Sequential):
    def __init__(self, depth: int = 12, **kwargs):
        super().__init__(*[TransformerEncoderBlock(**kwargs) for _ in range(depth)])
```

Head

- The last layer is a normal fully connect that gives the class probability. It first performs a basic mean over the whole sequence.

```
class ClassificationHead(nn.Sequential):
    def __init__(self, emb_size: int = 768, n_classes: int = 1000):
        super().__init__(
            Reduce('b n e -> b e', reduction='mean'),
            nn.LayerNorm(emb_size),
            nn.Linear(emb_size, n_classes))
```

1 ViT (Visual Transformer) in PyTorch

[출처] https://towardsdatascience.com/implementing-visualttransformer-in-pytorch-184f9f16f632
[Code] https://github.com/FrancescoSaverioZuppichini/ViT

3. Vi(sual) T(rasnformer)

- We can compose PatchEmbedding, TransformerEncoder and ClassificationHead to create the final ViT architecture.

```
class ViT(nn.Sequential):
    def __init__(self,
                 in_channels: int = 3,
                 patch_size: int = 16,
                 emb_size: int = 768,
                 img_size: int = 224,
                 depth: int = 12,
                 n_classes: int = 1000,
                 **kwargs):
        super().__init__(
            PatchEmbedding(in_channels, patch_size, emb_size, img_size),
            TransformerEncoder(depth, emb_size=emb_size, **kwargs),
            ClassificationHead(emb_size, n_classes)
        )
```

- We can use torchsummary to check the number of parameters

```
summary(ViT(), (3, 224, 224), device='cpu')
```

1 ViT (Visual Transformer) in PyTorch

[출처] https://towardsdatascience.com/implementing-visualttransformer-in-pytorch-184f9f16f632
[Code] https://github.com/FrancescoSaverioZuppichini/ViT

3. Vi(sual) T(rasnformer)

Layer (type)	Output Shape	Param #
Conv2d-1	[-1, 768, 14, 14]	590,592
Rearrange-2	[-1, 196, 768]	0
PatchEmbedding-3	[-1, 197, 768]	0
LayerNorm-4	[-1, 197, 768]	1,536
Linear-5	[-1, 197, 768]	590,592
Linear-6	[-1, 197, 768]	590,592
Linear-7	[-1, 197, 768]	590,592
Dropout-8	[-1, 8, 197, 197]	0
Linear-9	[-1, 197, 768]	590,592
MultiHeadAttention-10	[-1, 197, 768]	0
Dropout-11	[-1, 197, 768]	0
ResidualAdd-12	[-1, 197, 768]	0
LayerNorm-13	[-1, 197, 768]	1,536
Linear-14	[-1, 197, 3072]	2,362,368
GELU-15	[-1, 197, 3072]	0
Dropout-16	[-1, 197, 3072]	0
Linear-17	[-1, 197, 768]	2,360,064
Dropout-18	[-1, 197, 768]	0
ResidualAdd-19	[-1, 197, 768]	0
LayerNorm-20	[-1, 197, 768]	1,536
Linear-21	[-1, 197, 768]	590,592
Linear-22	[-1, 197, 768]	590,592
Linear-23	[-1, 197, 768]	590,592
Dropout-24	[-1, 8, 197, 197]	0
Linear-25	[-1, 197, 768]	590,592
MultiHeadAttention-26	[-1, 197, 768]	0
Dropout-27	[-1, 197, 768]	0
ResidualAdd-28	[-1, 197, 768]	0
LayerNorm-29	[-1, 197, 768]	1,536
Linear-30	[-1, 197, 3072]	2,362,368
GELU-31	[-1, 197, 3072]	0
Dropout-32	[-1, 197, 3072]	0
Linear-33	[-1, 197, 768]	2,360,064
Dropout-34	[-1, 197, 768]	0
ResidualAdd-35	[-1, 197, 768]	0
LayerNorm-36	[-1, 197, 768]	1,536
Linear-37	[-1, 197, 768]	590,592
Linear-38	[-1, 197, 768]	590,592
Linear-39	[-1, 197, 768]	590,592
Dropout-40	[-1, 8, 197, 197]	0
Linear-41	[-1, 197, 768]	590,592
MultiHeadAttention-42	[-1, 197, 768]	0
Dropout-43	[-1, 197, 768]	0
ResidualAdd-44	[-1, 197, 768]	0
LayerNorm-45	[-1, 197, 768]	1,536
Linear-46	[-1, 197, 3072]	2,362,368
GELU-47	[-1, 197, 3072]	0
Dropout-48	[-1, 197, 3072]	0
Linear-49	[-1, 197, 768]	2,360,064
Dropout-50	[-1, 197, 768]	0
ResidualAdd-51	[-1, 197, 768]	0
LayerNorm-52	[-1, 197, 768]	1,536
Linear-53	[-1, 197, 768]	590,592
Linear-54	[-1, 197, 768]	590,592
Linear-55	[-1, 197, 768]	590,592
Dropout-56	[-1, 8, 197, 197]	0
Linear-57	[-1, 197, 768]	590,592
MultiHeadAttention-58	[-1, 197, 768]	0
Dropout-59	[-1, 197, 768]	0
ResidualAdd-60	[-1, 197, 768]	0
LayerNorm-61	[-1, 197, 768]	1,536
Linear-62	[-1, 197, 3072]	2,362,368
GELU-63	[-1, 197, 3072]	0
Dropout-64	[-1, 197, 3072]	0
Linear-65	[-1, 197, 768]	2,360,064
Dropout-66	[-1, 197, 768]	0
ResidualAdd-67	[-1, 197, 768]	0
LayerNorm-68	[-1, 197, 768]	1,536
Linear-69	[-1, 197, 768]	590,592
Linear-70	[-1, 197, 768]	590,592
Linear-71	[-1, 197, 768]	590,592
Dropout-72	[-1, 8, 197, 197]	0
Linear-73	[-1, 197, 768]	590,592
MultiHeadAttention-74	[-1, 197, 768]	0
Dropout-75	[-1, 197, 768]	0
ResidualAdd-76	[-1, 197, 768]	0
LayerNorm-77	[-1, 197, 768]	1,536
Linear-78	[-1, 197, 3072]	2,362,368
GELU-79	[-1, 197, 3072]	0
Dropout-80	[-1, 197, 3072]	0
Linear-81	[-1, 197, 768]	2,360,064
Dropout-82	[-1, 197, 768]	0
ResidualAdd-83	[-1, 197, 768]	0
LayerNorm-84	[-1, 197, 768]	1,536
Linear-85	[-1, 197, 768]	590,592
Linear-86	[-1, 197, 768]	590,592
Linear-87	[-1, 197, 768]	590,592
Dropout-88	[-1, 8, 197, 197]	0
Linear-89	[-1, 197, 768]	590,592

1 ViT (Visual Transformer) in PyTorch

[출처] https://towardsdatascience.com/implementing-visualttransformer-in-pytorch-184f9f16f632
[Code] https://github.com/FrancescoSaverioZuppichini/ViT

3. Vi(sual) T(rasnformer)

MultiHeadAttention-90	[-1, 197, 768]	0	MultiHeadAttention-122	[-1, 197, 768]	0	MultiHeadAttention-154	[-1, 197, 768]	0
Dropout-91	[-1, 197, 768]	0	Dropout-123	[-1, 197, 768]	0	Dropout-155	[-1, 197, 768]	0
ResidualAdd-92	[-1, 197, 768]	0	ResidualAdd-124	[-1, 197, 768]	0	ResidualAdd-156	[-1, 197, 768]	0
LayerNorm-93	[-1, 197, 768]	1,536	LayerNorm-125	[-1, 197, 768]	1,536	LayerNorm-157	[-1, 197, 768]	1,536
Linear-94	[-1, 197, 3072]	2,362,368	Linear-126	[-1, 197, 3072]	2,362,368	Linear-158	[-1, 197, 3072]	2,362,368
GELU-95	[-1, 197, 3072]	0	GELU-127	[-1, 197, 3072]	0	GELU-159	[-1, 197, 3072]	0
Dropout-96	[-1, 197, 3072]	0	Dropout-128	[-1, 197, 3072]	0	Dropout-160	[-1, 197, 3072]	0
Linear-97	[-1, 197, 768]	2,360,064	Linear-129	[-1, 197, 768]	2,360,064	Linear-161	[-1, 197, 768]	2,360,064
Dropout-98	[-1, 197, 768]	0	Dropout-130	[-1, 197, 768]	0	Dropout-162	[-1, 197, 768]	0
ResidualAdd-99	[-1, 197, 768]	0	ResidualAdd-131	[-1, 197, 768]	0	ResidualAdd-163	[-1, 197, 768]	0
LayerNorm-100	[-1, 197, 768]	1,536	LayerNorm-132	[-1, 197, 768]	1,536	LayerNorm-164	[-1, 197, 768]	1,536
Linear-101	[-1, 197, 768]	590,592	Linear-133	[-1, 197, 768]	590,592	Linear-165	[-1, 197, 768]	590,592
Linear-102	[-1, 197, 768]	590,592	Linear-134	[-1, 197, 768]	590,592	Linear-166	[-1, 197, 768]	590,592
Linear-103	[-1, 197, 768]	590,592	Linear-135	[-1, 197, 768]	590,592	Linear-167	[-1, 197, 768]	590,592
Dropout-104	[-1, 8, 197, 197]	0	Dropout-136	[-1, 8, 197, 197]	0	Dropout-168	[-1, 8, 197, 197]	0
Linear-105	[-1, 197, 768]	590,592	Linear-137	[-1, 197, 768]	590,592	Linear-169	[-1, 197, 768]	590,592
MultiHeadAttention-106	[-1, 197, 768]	0	MultiHeadAttention-138	[-1, 197, 768]	0	MultiHeadAttention-170	[-1, 197, 768]	0
Dropout-107	[-1, 197, 768]	0	Dropout-139	[-1, 197, 768]	0	Dropout-171	[-1, 197, 768]	0
ResidualAdd-108	[-1, 197, 768]	0	ResidualAdd-140	[-1, 197, 768]	0	ResidualAdd-172	[-1, 197, 768]	0
LayerNorm-109	[-1, 197, 768]	1,536	LayerNorm-141	[-1, 197, 768]	1,536	LayerNorm-173	[-1, 197, 768]	1,536
Linear-110	[-1, 197, 3072]	2,362,368	Linear-142	[-1, 197, 3072]	2,362,368	Linear-174	[-1, 197, 3072]	2,362,368
GELU-111	[-1, 197, 3072]	0	GELU-143	[-1, 197, 3072]	0	GELU-175	[-1, 197, 3072]	0
Dropout-112	[-1, 197, 3072]	0	Dropout-144	[-1, 197, 3072]	0	Dropout-176	[-1, 197, 3072]	0
Linear-113	[-1, 197, 768]	2,360,064	Linear-145	[-1, 197, 768]	2,360,064	Linear-177	[-1, 197, 768]	2,360,064
Dropout-114	[-1, 197, 768]	0	Dropout-146	[-1, 197, 768]	0	Dropout-178	[-1, 197, 768]	0
ResidualAdd-115	[-1, 197, 768]	0	ResidualAdd-147	[-1, 197, 768]	0	ResidualAdd-179	[-1, 197, 768]	0
LayerNorm-116	[-1, 197, 768]	1,536	LayerNorm-148	[-1, 197, 768]	1,536	LayerNorm-180	[-1, 197, 768]	1,536
Linear-117	[-1, 197, 768]	590,592	Linear-149	[-1, 197, 768]	590,592	Linear-181	[-1, 197, 768]	590,592
Linear-118	[-1, 197, 768]	590,592	Linear-150	[-1, 197, 768]	590,592	Linear-182	[-1, 197, 768]	590,592
Linear-119	[-1, 197, 768]	590,592	Linear-151	[-1, 197, 768]	590,592	Linear-183	[-1, 197, 768]	590,592
Dropout-120	[-1, 8, 197, 197]	0	Dropout-152	[-1, 8, 197, 197]	0	Dropout-184	[-1, 8, 197, 197]	0
Linear-121	[-1, 197, 768]	590,592	Linear-153	[-1, 197, 768]	590,592	Linear-185	[-1, 197, 768]	590,592

1 ViT (Visual Transformer) in PyTorch

[출처] https://towardsdatascience.com/implementing-visualttransformer-in-pytorch-184f9f16f632
[Code] https://github.com/FrancescoSaverioZuppichini/ViT

3. Vi(sual) T(rasnformer)

```
MultiHeadAttention-186            [-1, 197, 768]               0
         Dropout-187            [-1, 197, 768]               0
     ResidualAdd-188            [-1, 197, 768]               0
       LayerNorm-189            [-1, 197, 768]           1,536
          Linear-190           [-1, 197, 3072]       2,362,368
            GELU-191           [-1, 197, 3072]               0
         Dropout-192           [-1, 197, 3072]               0
          Linear-193            [-1, 197, 768]       2,360,064
         Dropout-194            [-1, 197, 768]               0
     ResidualAdd-195            [-1, 197, 768]               0
          Reduce-196                 [-1, 768]               0
       LayerNorm-197                 [-1, 768]           1,536
          Linear-198                [-1, 1000]         769,000
================================================================
Total params: 86,415,592
Trainable params: 86,415,592
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.57
Forward/backward pass size (MB): 364.33
Params size (MB): 329.65
Estimated Total Size (MB): 694.56
```

1 ViT (Visual Transformer) in PyTorch

[출처] https://gaussian37.github.io/dl-concept-vit/

- Another code

```
import torch
import torch.nn as nn

class LinearProjection(nn.Module):

    def __init__(self, patch_vec_size, num_patches, latent_vec_dim, drop_rate):
        super().__init__()
        self.linear_proj = nn.Linear(patch_vec_size, latent_vec_dim)
        self.cls_token = nn.Parameter(torch.randn(1, latent_vec_dim))
        self.pos_embedding = nn.Parameter(torch.randn(1, num_patches+1,
latent_vec_dim))
        self.dropout = nn.Dropout(drop_rate)

    def forward(self, x):
        batch_size = x.size(0)
        x = torch.cat([self.cls_token.repeat(batch_size, 1, 1), self.linear_proj(x)],
dim=1)
        x += self.pos_embedding
        x = self.dropout(x)
        return x
```

```
class MultiheadedSelfAttention(nn.Module):
    def __init__(self, latent_vec_dim, num_heads, drop_rate):
        super().__init__()
        device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
        self.num_heads = num_heads
        self.latent_vec_dim = latent_vec_dim
        self.head_dim = int(latent_vec_dim / num_heads)
        self.query = nn.Linear(latent_vec_dim, latent_vec_dim)
        self.key = nn.Linear(latent_vec_dim, latent_vec_dim)
        self.value = nn.Linear(latent_vec_dim, latent_vec_dim)
        self.scale = torch.sqrt(latent_vec_dim*torch.ones(1)).to(device)
        self.dropout = nn.Dropout(drop_rate)

    def forward(self, x):
        batch_size = x.size(0)
        q = self.query(x)
        k = self.key(x)
        v = self.value(x)
        q = q.view(batch_size, -1, self.num_heads, self.head_dim).permute(0,2,1,3)
        k = k.view(batch_size, -1, self.num_heads, self.head_dim).permute(0,2,3,1) # k.t
        v = v.view(batch_size, -1, self.num_heads, self.head_dim).permute(0,2,1,3)
        attention = torch.softmax(q @ k / self.scale, dim=-1)
        x = self.dropout(attention) @ v
        x = x.permute(0,2,1,3).reshape(batch_size, -1, self.latent_vec_dim)

        return x, attention
```

1 ViT (Visual Transformer) in PyTorch

[출처] https://gaussian37.github.io/dl-concept-vit/

```
class TFencoderLayer(nn.Module):
    def __init__(self, latent_vec_dim, num_heads, mlp_hidden_dim, drop_rate):
        super().__init__()
        self.ln1 = nn.LayerNorm(latent_vec_dim)
        self.ln2 = nn.LayerNorm(latent_vec_dim)
        self.msa = MultiheadedSelfAttention(latent_vec_dim=latent_vec_dim,
num_heads=num_heads, drop_rate=drop_rate)
        self.dropout = nn.Dropout(drop_rate)
        self.mlp = nn.Sequential(nn.Linear(latent_vec_dim, mlp_hidden_dim),
                                 nn.GELU(), nn.Dropout(drop_rate),
                                 nn.Linear(mlp_hidden_dim, latent_vec_dim),
                                 nn.Dropout(drop_rate))

    def forward(self, x):
        z = self.ln1(x)
        z, att = self.msa(z)
        z = self.dropout(z)
        x = x + z
        z = self.ln2(x)
        z = self.mlp(z)
        x = x + z

        return x, att
```

```
class VisionTransformer(nn.Module):
    def __init__(self, patch_vec_size, num_patches, latent_vec_dim, num_heads,
mlp_hidden_dim, drop_rate, num_layers, num_classes):
        super().__init__()
        self.patchembedding = LinearProjection(patch_vec_size=patch_vec_size,
num_patches=num_patches,
                                               latent_vec_dim=latent_vec_dim,
drop_rate=drop_rate)
        self.transformer =
nn.ModuleList([TFencoderLayer(latent_vec_dim=latent_vec_dim,
num_heads=num_heads, mlp_hidden_dim=mlp_hidden_dim, drop_rate=drop_rate)
                                       for _ in range(num_layers)])

        self.mlp_head = nn.Sequential(nn.LayerNorm(latent_vec_dim),
nn.Linear(latent_vec_dim, num_classes))

    def forward(self, x):
        att_list = []
        x = self.patchembedding(x)
        for layer in self.transformer:
            x, att = layer(x)
            att_list.append(att)
        x = self.mlp_head(x[:,0])

        return x, att_list
```

2 DeiT (Data-efficient Image Transformers)

[Facebook AI]
Hugo Touvron, Matthieu Cord, Matthijs Douze, Francisco Massa, Alexandre Sablayrolles and Hervé Jégou,
"Training data-efficient image transformers & distillation through attention," arXiv 2021

facebookresearch/deit
https://github.com/facebookresearch/deit

Optimizing Vision Transformer Model for Deployment
https://pytorch.org/tutorials/beginner/vt_tutorial.html

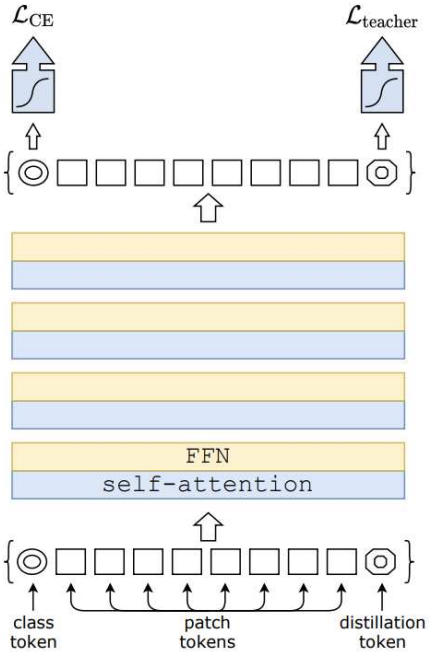
Abstract

- Recently, **neural networks purely based on attention** were shown to address **image classification**.
- These high performing **vision transformers** are **pre-trained with hundreds of millions of images** using a large infrastructure, thereby limiting their adoption.

❖ This work produces competitive **convolution-free transformers by training on Imagenet only**.
- **Train them on a single computer in less than 3 days.**
- **Our reference vision transformer (86M parameters) achieves top-1 accuracy of 83.1% (single-crop) on ImageNet with no external data.**
- Introduce a **teacher-student strategy specific to transformers**. It **relies on a distillation token ensuring that the student learns from the teacher through attention.**
- We show the interest of this **token-based distillation**, especially when using a convnet as a teacher. This leads us to report results competitive with convnets for both Imagenet (where we obtain up to 85.2% accuracy) and when transferring to other tasks.

2 DeiT (Data-efficient Image Transformers)

- **ViT** : Pre-trained with 100M images (large infrastructure) - limiting their adoption.
  - Excellent results trained with JFT-300M, 300M images
  - They concluded that transformers *"do not generalize well when trained on insufficient amounts of data"*.
- **DeiT** : Training on Imagenet only, **Single node with 4GPU in 3 days**
  - 86M parameters : top-1 accuracy 83.1% (single-crop) on ImageNet without external data
  - **Teacher-student strategy** : a **distillation token** ensuring that the student learns from the teach through attention



Model Zoo

We provide baseline DeiT models pretrained on ImageNet 2012.

name	acc@1	acc@5	#params	url
DeiT-tiny	72.2	91.1	5M	model
DeiT-small	79.9	95.0	22M	model
DeiT-base	81.8	95.6	86M	model
DeiT-tiny distilled	74.5	91.9	6M	model
DeiT-small distilled	81.2	95.4	22M	model
DeiT-base distilled	83.4	96.5	87M	model
DeiT-base 384	82.9	96.2	87M	model
DeiT-base distilled 384 (1000 epochs)	85.2	97.2	88M	model
CaiT-S24 distilled 384	85.1	97.3	47M	model
CaiT-M48 distilled 448	86.5	97.7	356M	model

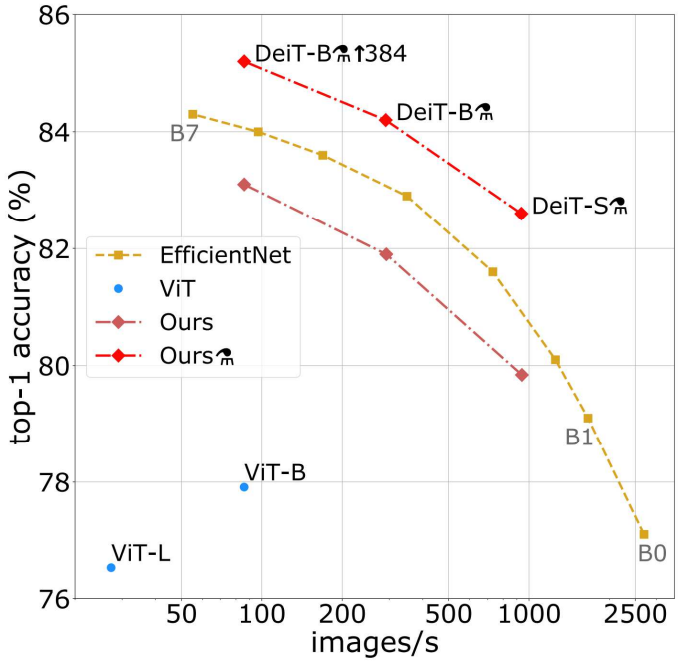


Figure 1: Throughput and accuracy on Imagenet of our methods compared to EfficientNets, trained on Imagenet1k only. The throughput is measured as **the number of images processed per second on a V100 GPU**. DeiT-B is identical to VIT-B, but the training is more adapted to a data-starving regime. It is learned in a few days on one machine. The symbol ⚗ refers to models trained with our transformer-specific distillation. See Table 5 for details and more models.

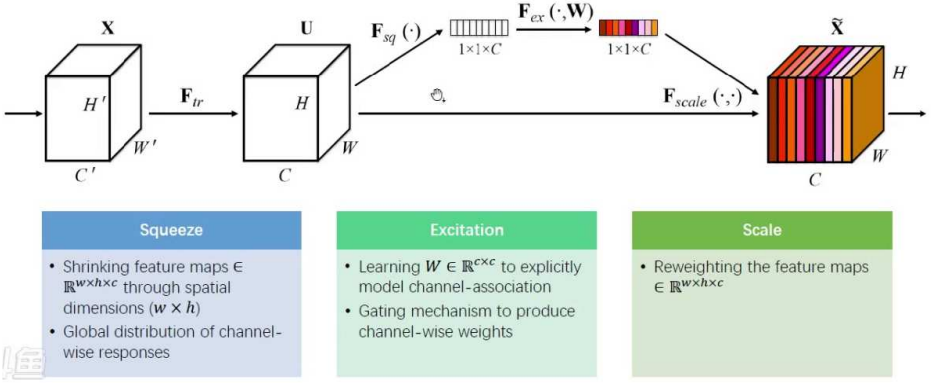
2 DeiT (Data-efficient Image Transformers)

Hugo Touvron, et al. "Training data-efficient image transformers & distillation through attention," arXiv 2021

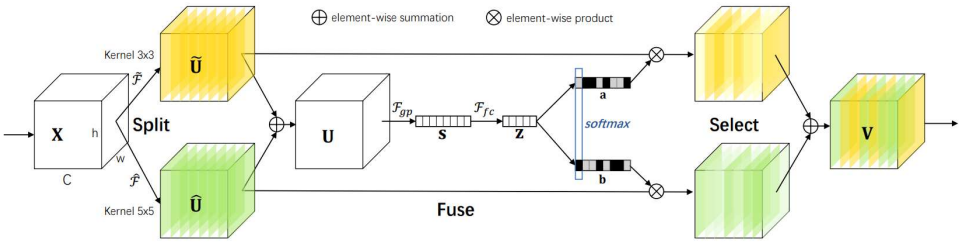
Related Works : Transformer Architecture

Transformer

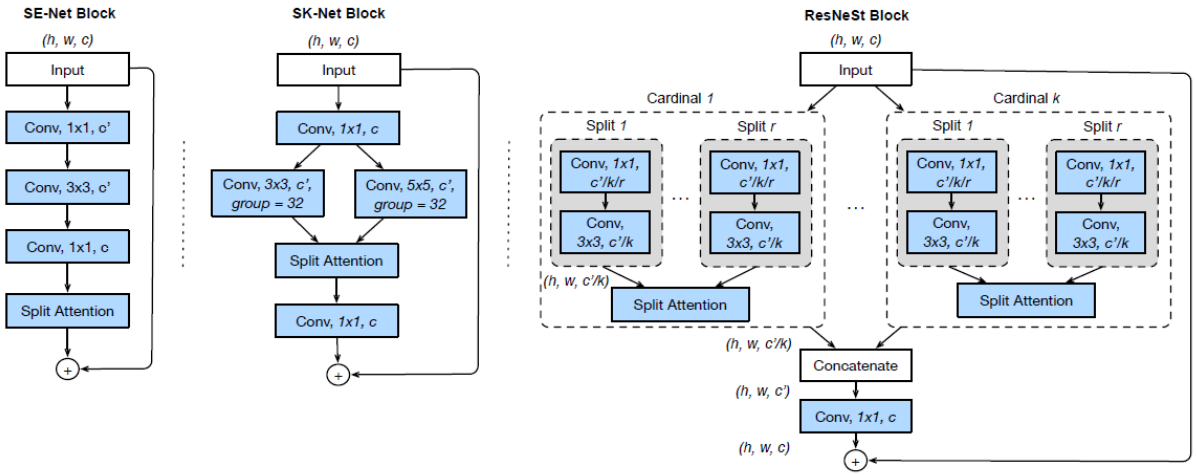
- Introduce by Vaswani et al. (Attention is All you need, 2017) for machine translation
- Currently the reference model for all NLP tasks
- Many improvements of convnets for image classification are inspired by transformers. Ex) Squeeze & Excitation, Selective Kernel, Split-attention networks exploit mechanism akin to transformers self-attention (SA) mechanism.



Squeeze-and-excitation (SE block) (2017)



Selective-Kernel Networks (SK-net) (2019)



Split-Attention Networks (2020)

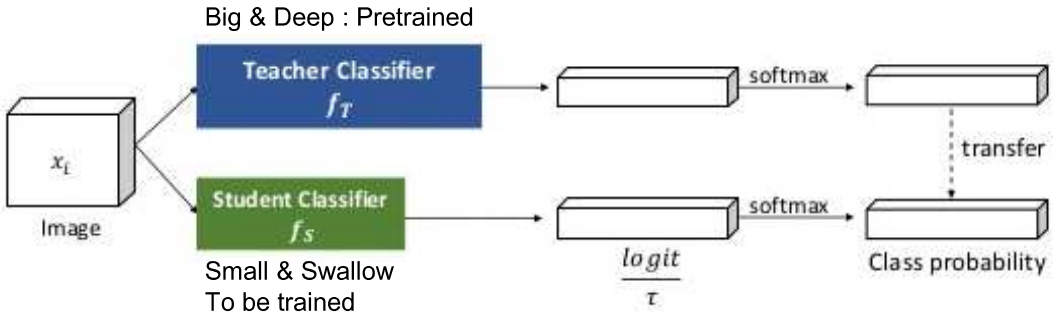
2 DeiT (Data-efficient Image Transformers)

Hugo Touvron, et al. "Training data-efficient image transformers & distillation through attention," arXiv 2021

Related Works : Knowledge Distillation (KD)

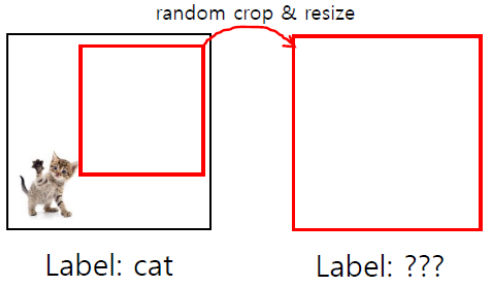
- Introduce Hinton et al. (“Distilling the Knowledge in a neural network”, 2015)
- ✓ **KD refers to the training paradigm in which a student model** leverages **"soft" labels** coming from **a strong teacher network** → This is the **output vector of teacher's softmax function**, rather than just the maximum of scores, which give **"hard" labels**

Objective: $\sum_{x_i \in \mathcal{X}} \mathrm{KL}\left(\mathrm{softmax}\left(\frac{f_T(x_i)}{\tau}\right), \mathrm{softmax}\left(\frac{f_S(x_i)}{\tau}\right)\right)$



[출처] slideshare, Wonpyo Park, "Relational knowledge distillation"
[출처] PR12 Paper Review, Jinwon Lee, PR-297 DeiT

- By Wei et al.[54] (“Circumventing Outliers of AutoAugment with Knowledge Distillation”, 2020)
- The teacher's supervision takes into account the effects of the data augmentation, which sometimes causes a misalignment between the real label and the image.



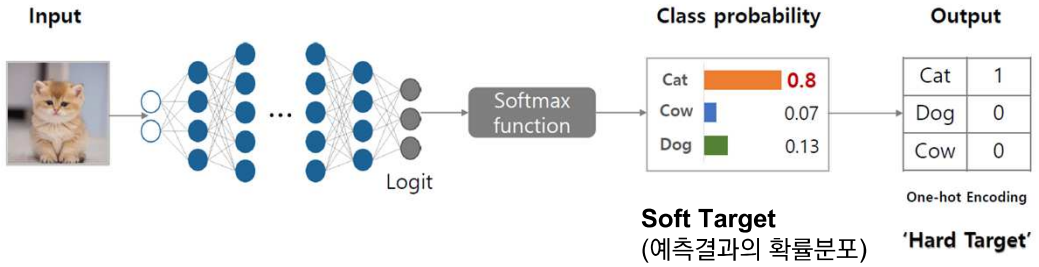
- Abnar et al. (Transferring Inductive Biases through Knowledge Distillation, 2020)
- KD can transfer inductive biases in a soft way in a student model using a teacher model where they would be incorporated in a hard way. → Useful to induce biases due to convolutions in a transformer model by **using a convolutional model as a teacher**.

2 DeiT (Data-efficient Image Transformers)

Hugo Touvron, et al. "Training data-efficient image transformers & distillation through attention," arXiv 2021

Related Works : Knowledge Distillation (KD)

Knowledge : Soft Target

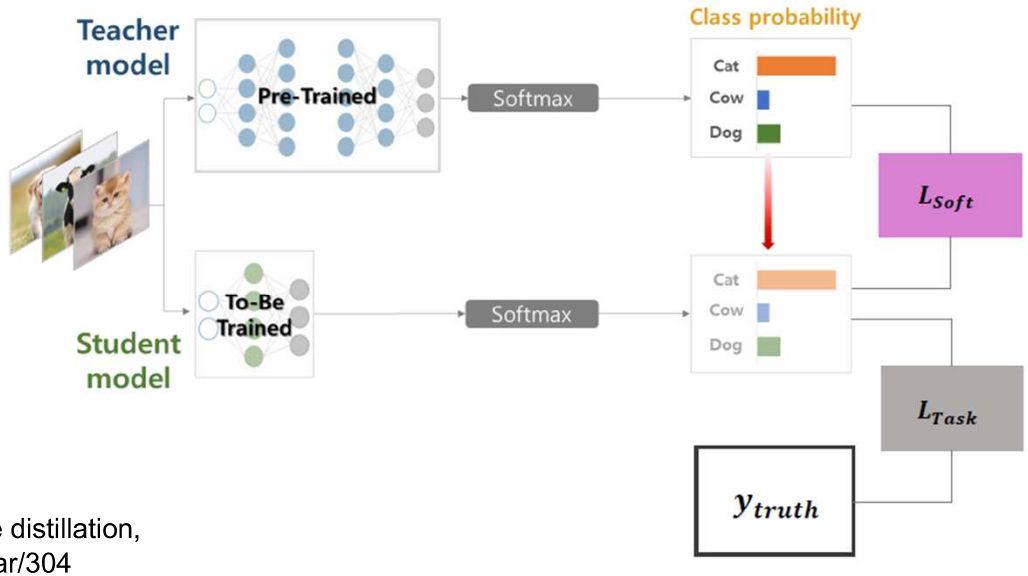


$$Softmax(z_i) = \frac{\exp(z_i)}{\sum_j \exp(z_j)} \rightarrow Softmax(z_i) = \frac{\exp(z_i/\tau)}{\sum_j \exp(z_j/\tau)}$$

τ ($Temperature$): Scaling 역할의 하이퍼 파라미터
- τ = 1일 때, 기존 softmax function과 동일
- τ클수록, 더 soft한 확률분포

Distillation Method :
Offline distillation
(Response-based)

Response-Based
Feature-Based
Relation-Based



- $f_T(x_i)$: Teacher 모델의 logit 값
- $f_T(x_i)$: Student 모델의 logit 값
- τ :Scaling 역할의 하이퍼 파라미터

$$L_{Soft} = \sum_{x_i \in X} KL(softmax(\frac{f_T(x_i)}{\tau}), softmax(\frac{f_s(x_i)}{\tau}))$$

$$L_{Task} = CrossEntropy\ (softmax(f_s(x_i)), y_{truth})$$

$$Student\ L_{Total} = L_{Task} + \lambda \cdot L_{Soft}$$

[출처] 황하은, Introduction to knowledge distillation,
http://dmqm.korea.ac.kr/activity/seminar/304

2 DeiT (Data-efficient Image Transformers)

Vision Transformer - Same Architecture as ViT

Multi-head Self-Attention Layers (MSA)

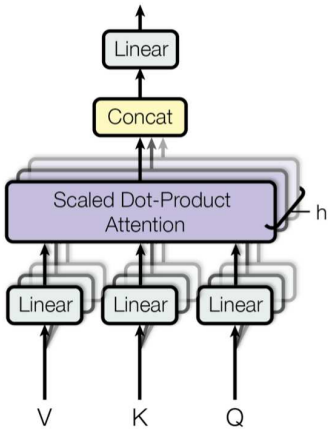
Multi-head Self Attention layers (MSA). The attention mechanism is based on a trainable associative memory with (key, value) vector pairs. A *query* vector $q \in \mathbb{R}^d$ is matched against a set of k *key* vectors (packed together into a matrix $K \in \mathbb{R}^{k \times d}$) using inner products. These inner products are then scaled and normalized with a softmax function to obtain k weights. The output of the attention is the weighted sum of a set of k *value* vectors (packed into $V \in \mathbb{R}^{k \times d}$). For a sequence of N query vectors (packed into $Q \in \mathbb{R}^{N \times d}$), it produces an output matrix (of size $N \times d$):

$$\text{Attention}(Q, K, V) = \text{Softmax}(QK^\top/\sqrt{d})V, \quad (1)$$

where the Softmax function is applied over each row of the input matrix and the $\sqrt{d}$ term provides appropriate normalization.

In [52], a Self-attention layer is proposed. Query, key and values matrices are themselves computed from a sequence of N input vectors (packed into $X \in \mathbb{R}^{N \times D}$): $Q = XW_\text{Q}$, $K = XW_\text{K}$, $V = XW_\text{V}$, using linear transformations $W_\text{Q}, W_\text{K}, W_\text{V}$ with the constraint $k = N$, meaning that the attention is in between all the input vectors.
Finally, Multi-head self-attention layer (MSA) is defined by considering h attention "heads", *ie* h self-attention functions applied to the input. Each head provides a sequence of size $N \times d$. These h sequences are rearranged into a $N \times dh$ sequence that is reprojected by a linear layer into $N \times D$.

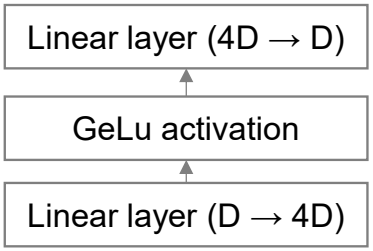
Hugo Touvron, et al. "Training data-efficient image transformers & distillation through attention," arXiv 2021



PxP : 16x16
N : 224/16 x 224/16 = 14x14
D : 16x16x3 = 768
h : 8, d = 96

Transformer block for images

- We add a **Feed-Forward Network (FFN)** on **top of MSA layer.**



D : 16x16x3 = 768

- Both MSA and FFN are operating as **residual operation** (thanks to skip-connection), and with a **layer normalization.**

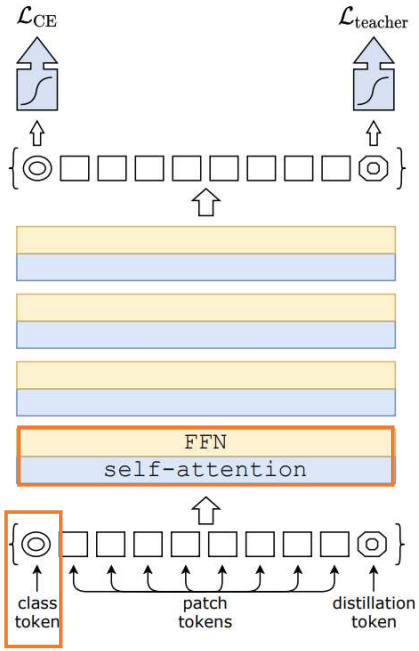
2 DeiT (Data-efficient Image Transformers)

Hugo Touvron, et al. "Training data-efficient image transformers & distillation through attention," arXiv 2021

Vision Transformer - Same Architecture as ViT

Class token

- Trainable vector, appended to the patch token before the first layer, → goes through the transformer layer → projected with a linear layer to predict the class.
- Transformer process batches of $(N+1)$ tokens of dimension D, of which only the class token is used to predict the output.
- Forces the self-attention to spread information between the patch tokens and the class token.
- At training time, the **supervision signal** comes only from the **class embedding**, while the **patch tokens** are the **model's only variable output**.



Fixing the positional encoding across resolution

- Tourvron et al. (*Fixefficientnet*, 2020) show that it is desirable to use a lower training resolution and fine-tune the network the larger resolution
  - ✓ Speed up the full training and improves the accuracy under prevailing data augmentation schemes.

→ When increasing the resolution of an input image, patch size does not change, therefore the number of input patches(N) does change. One need to adapt the positional embeddings.

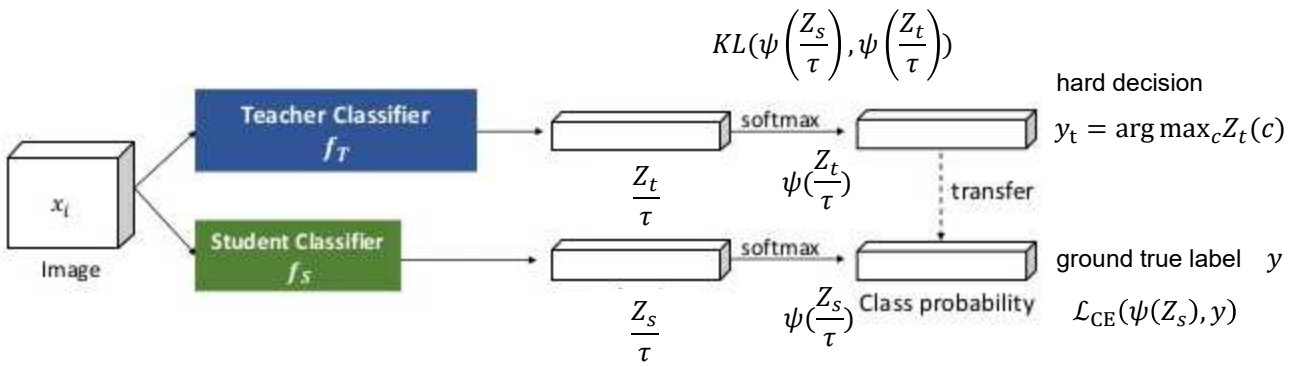
→ Dosovitaskiy et al. [15] (ViT) interpolate the positional encoding when changing the resolution. → Work with the subsequent fine-tuning stage.

2 DeiT (Data-efficient Image Transformers)

Hugo Touvron, et al. "Training data-efficient image transformers & distillation through attention," arXiv 2021

Distillation through Attention

- Assume that **a strong image classifier as a teacher model**
- **Hard distillation** vs **Soft distillation**,
- **Classical distillation** vs **Distillation token**



1) Soft distillation

- Minimizes the Kullback-Leiber divergence between the teacher's softmax and the student's softmax.
- Distillation objective

$$\mathcal{L}_{global} = (1-\lambda)\mathcal{L}_{CE}(\psi(Z_s), y) + \lambda\tau^2 KL(\psi(\frac{Z_s}{\tau}), \psi(\frac{Z_t}{\tau}))$$

- ✓ Z_t , Z_s : the logits of teacher and student models
- ✓ τ : the temperature for the distillation
- ✓ λ : the coefficient balancing the KL divergence loss ($KL()$) and the cross-entropy ($\mathcal{L}_{CE}$) on ground truth labels y
- ✓ ψ : softmax ft.

2) Hard distillation

- Take the hard decision of the teacher ($y_t = \arg\max_c Z_t(c)$) as a true label
- Hard-label distillation based objective

$$\mathcal{L}^{hardDistill}_{global} = \frac{1}{2}\mathcal{L}_{CE}(\psi(Z_s), y) + \frac{1}{2}\mathcal{L}_{CE}(\psi(Z_s), y_t)$$

- ✓ For a given image, the hard label associated with the teacher may change depending on the specific data augmentation.
- ✓ The teacher prediction y_t plays the same role as the true label y.

- The hard-label can be converted into soft labels with label smoothing, where the true label is considered to have a probability $1-\epsilon$ (ϵ=0.1) and the remaining ϵ is shared across the remaining classes.

2 DeiT (Data-efficient Image Transformers)

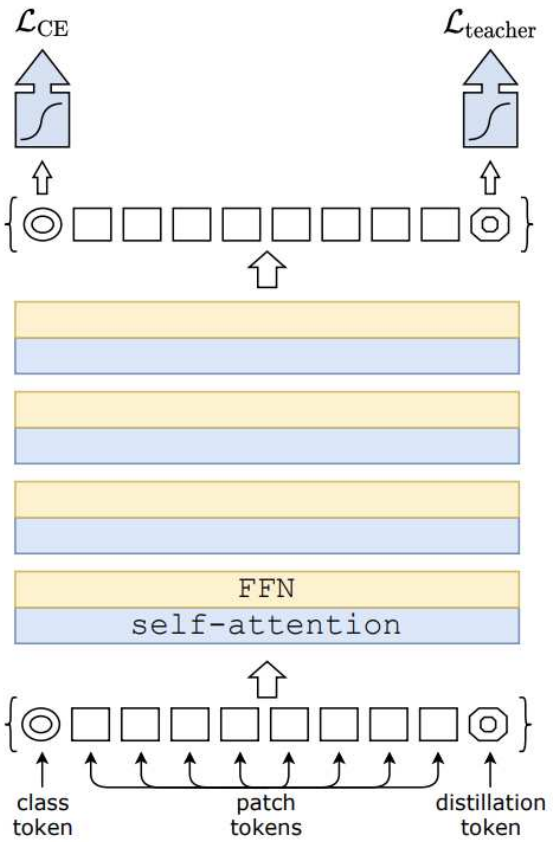
Hugo Touvron, et al. "Training data-efficient image transformers & distillation through attention," arXiv 2021

Distillation through Attention

3) Distillation token

- Add a **new distillation token** to class token/patch tokens.
- It **interacts with the class and patch tokens through the self-attention layers.**
- This distillation token is employed in a similar fashion as the class token, except that on output of the network, its objective is **to reproduce the (hard) label predicted by the teacher, instead of true label**. (The target objective is given by the distillation component of the loss.)
- Distillation embedding allows the model to learn from the output of the teacher, while remaining complementary to the class embedding.
- **Both the class and distillation tokens input to the transformers are learned by back-propagation.**
- The learned class/distillation tokens converge towards different vectors; the average cosine similarity between two tokens equal to 0.06. → at the last layer, their similarity equal to 0.93.

$$\text{similarity} = \cos(\theta) = \frac{A \cdot B}{\|A\|\|B\|} = \frac{\sum_{i=1}^{n} A_i \times B_i}{\sqrt{\sum_{i=1}^{n} (A_i)^2} \times \sqrt{\sum_{i=1}^{n} (B_i)^2}}$$



Our distillation procedure:

2 DeiT (Data-efficient Image Transformers)

Hugo Touvron, et al. "Training data-efficient image transformers & distillation through attention," arXiv 2021

Distillation through Attention

4) Fine-tuning with distillation

- Use **both the true label and teacher prediction** during the fine-tuning stage at higher resolution.
- Use **a teacher with the same target resolution**, typically **obtained from the lower-resolution teacher**.

5) Classification with our approach: Joint classifier

- At test time, both *the class and distillation embeddings* (produced by transformer) are associated with *linear classifiers* and able to infer the image label.
- Our referent method is the late fusion of these two separate heads, for which we add the softmax output by two classifiers to make the prediction.

2 DeiT (Data-efficient Image Transformers)

Hugo Touvron, et al. "Training data-efficient image transformers & distillation through attention," arXiv 2021

[출처] PR12 Paper Review, Jinwon Lee, PR-297 DeiT

Experiments

Transformer Models

- Architecture is identical to ViT with no convolution (ViT-B = DeiT-B)
- Only differences : Training strategy and Distillation token

- DeiT-B : Reference model (Same as ViT-B) → Parameters are fixed as D=768, h=12, d=D/h=64 (Keeping d=64)
- DeiT-B↑384 : Fine-tune DeiT at a larger resolution
- DeiT⚗ : DeiT with distillation (using distillation token)
- DeiT-S (Small), DeiT-Ti (Tiny) : Smaller models of DeiT

Model	ViT model	embedding dimension	#heads	#layers	#params	training resolution	throughput (im/sec)
DeiT-Ti	N/A	192	3	12	5M	224	2536
DeiT-S	N/A	384	6	12	22M	224	940
DeiT-B	ViT-B	768	12	12	86M	224	292

Table 1: Variants of our DeiT architecture. The larger model, DeiT-B, has the same architecture as the ViT-B [15]. The only parameters that vary across models are the embedding dimension and the number of heads, and we keep the dimension per head constant (equal to 64). Smaller models have a lower parameter count, and a faster throughput. The throughput is measured for images at resolution 224×224.

Distillation : Convnets teachers

- Using a **convnet teacher** gives better performance than using a transformer
- Due to the **inductive bias** inherited by the transformer through distillation

Teacher Models	acc.	Student: DeiT-B ⚗ pretrain	↑384
DeiT-B	81.8	81.9	83.1
RegNetY-4GF	80.0	82.7	83.6
RegNetY-8GF	81.7	82.7	83.8
RegNetY-12GF	82.4	83.1	84.1
RegNetY-16GF	82.9	83.1	84.2

- Default teacher is a RegNetY-16CF (84M parameter)

  https://paperswithcode.com/method/regnety

2 DeiT (Data-efficient Image Transformers)

Hugo Touvron, et al. "Training data-efficient image transformers & distillation through attention," arXiv 2021

[출처] PR12 Paper Review, Jinwon Lee, PR-297 DeiT

Experiments

Distillation : distillation methods

- Hard distillation significantly outperforms soft distillation for transformers, even when using only a class token.
- The classifier on the two tokens is significantly better than the independent class and distillation classifiers
- The distillation token gives slightly better results than the class token. It is more correlated to the convnets prediction.

method ↓	Supervision label	Supervision teacher	Test tokens class	Test tokens distil.	ImageNet top-1 (%) pretrain	ImageNet top-1 (%) finetune 384
DeiT– no distillation	✓		✓		81.8	83.1
DeiT– usual distillation	✓	soft	✓		81.8	83.1
DeiT– hard distillation	✓	hard	✓		83.0	84.0
DeiT⚗: class embedding	✓	hard	✓		83.0	84.1
DeiT⚗: distil. embedding	✓	hard		✓	83.1	84.2
DeiT⚗: class+distillation	✓	hard	✓	✓	83.4	84.2

Table 3: Distillation experiments on Imagenet with DeiT, 300 epochs of pretraining. We separately report the performance when classifying with only one of the class or distillation embeddings, and then with a classifier taking both of them as input. In the last row (class+distillation), the result correspond to the late fusion of the class and distillation classifiers.

Distillation : Agreement with the teacher & inductive bias

- Does it inherit existing inductive bias that would facilitate the training?
- Below table reports the fraction of sample classified differently for all classifier pairs, i.e. the rate of different decisions.
- The distilled model is more correlated to the convnet than a transformer learned from scratch.

	groundtruth	no distillation convnet	no distillation DeiT	DeiT⚗ student (of the convnet) class	DeiT⚗ student (of the convnet) distillation	DeiT⚗ student (of the convnet) DeiT⚗
groundtruth	0.000	0.171	0.182	0.170	0.169	0.166
convnet (RegNetY)	0.171	0.000	0.133	0.112	0.100	0.102
DeiT	0.182	0.133	0.000	0.109	0.110	0.107
DeiT⚗– class only	0.170	0.112	0.109	0.000	0.050	0.033
DeiT⚗– distil. only	0.169	0.100	0.110	0.050	0.000	0.019
DeiT⚗– class+distil.	0.166	0.102	0.107	0.033	0.019	0.000

Table 4: Disagreement analysis between convnet, image transformers and distillated transformers: We report the fraction of sample classified differently for all classifier pairs, i.e., the rate of different decisions. We include two models without distillation (a RegNetY and DeiT-B), so that we can compare how our distilled models and classification heads are correlated to these teachers.

2 DeiT (Data-efficient Image Transformers)

Hugo Touvron, et al. "Training data-efficient image transformers & distillation through attention," arXiv 2021

[출처] PR12 Paper Review, Jinwon Lee, PR-297 DeiT

Experiments

Distillation : Number of epochs

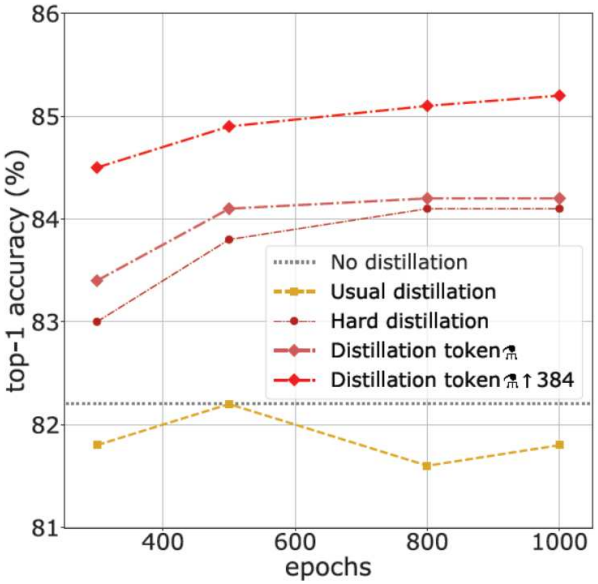
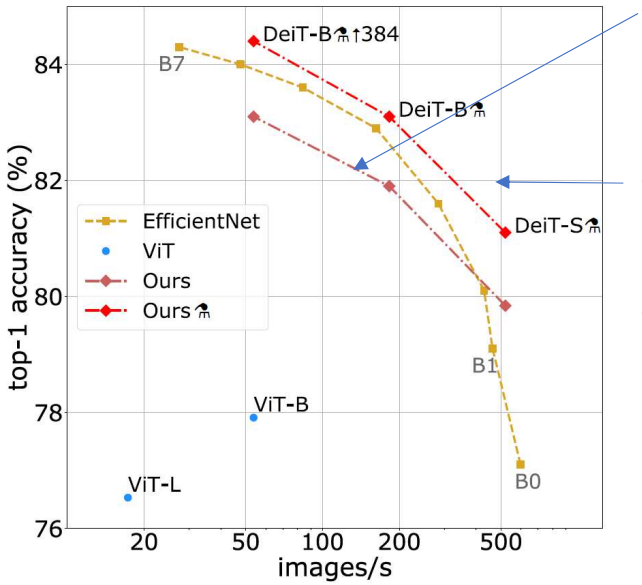


Figure 3: Distillation on ImageNet [42] with DeiT-B: performance as a function of the number of training epochs. We provide the performance without distillation (horizontal dotted line) as it saturates after 400 epochs.

With 300 epochs, our distilled network DeiT-B⚗ is already better than DeiT-B. But while for the latter the performance saturates with longer schedules, our distilled network clearly benefits from a longer training time.

Efficiency vs accuracy : Comparative study with convnets



- DeiT is slightly below EfficientNet, which shows that almost closed the gap between visual transformer and convnets when training with Imagenet only.
- These results are a major improvement (+6.3% top-1 in a comparable setting) over previous ViT models trained on Imagenet1k only.
- Furthermore, when DeiT benefits from the distillation from a relatively weaker RegNetY to produce DeiT⚗ , it outperforms EfficientNet.

2 DeiT (Data-efficient Image Transformers)

Hugo Touvron, et al. "Training data-efficient image transformers & distillation through attention," arXiv 2021

[출처] PR12 Paper Review, Jinwon Lee, PR-297 DeiT

Experiments

Efficiency vs accuracy : Comparative study with convnets

- Compared to EfficientNet, one can see that, for the same number of parameters, the convnet variants are much slower. This is because large matrix multiplication offers more opportunity for hardware optimization than small convolutions.

Network	#param.	image size	throughput (image/s)	ImNet top-1	Real top-1	V2 top-1
Convnets						
ResNet-18 [21]	12M	224^2	4458.4	69.8	77.3	57.1
ResNet-50 [21]	25M	224^2	1226.1	76.2	82.5	63.3
ResNet-101 [21]	45M	224^2	753.6	77.4	83.7	65.7
ResNet-152 [21]	60M	224^2	526.4	78.3	84.1	67.0
RegNetY-4GF [40]*	21M	224^2	1156.7	80.0	86.4	69.4
RegNetY-8GF [40]*	39M	224^2	591.6	81.7	87.4	70.8
RegNetY-16GF [40]*	84M	224^2	334.7	82.9	88.1	72.4
EfficientNet-B0 [48]	5M	224^2	2694.3	77.1	83.5	64.3
EfficientNet-B1 [48]	8M	240^2	1662.5	79.1	84.9	66.9
EfficientNet-B2 [48]	9M	260^2	1255.7	80.1	85.9	68.8
EfficientNet-B3 [48]	12M	300^2	732.1	81.6	86.8	70.6
EfficientNet-B4 [48]	19M	380^2	349.4	82.9	88.0	72.3
EfficientNet-B5 [48]	30M	456^2	169.1	83.6	88.3	73.6
EfficientNet-B6 [48]	43M	528^2	96.9	84.0	88.8	73.9
EfficientNet-B7 [48]	66M	600^2	55.1	84.3	-	-
EfficientNet-B5 RA [12]	30M	456^2	96.9	83.7	-	-
EfficientNet-B7 RA [12]	66M	600^2	55.1	84.7	-	-
KDforAA-B8	87M	800^2	25.2	85.8	-	-
Transformers						
ViT-B/16 [15]	86M	384^2	85.9	77.9	83.6	-
ViT-L/16 [15]	307M	384^2	27.3	76.5	82.2	-
DeiT-Ti	5M	224^2	2536.5	72.2	80.1	60.4
DeiT-S	22M	224^2	940.4	79.8	85.7	68.5
DeiT-B	86M	224^2	292.3	81.8	86.7	71.5
DeiT-B↑384	86M	384^2	85.9	83.1	87.7	72.4
DeiT-Ti⚗	6M	224^2	2529.5	74.5	82.1	62.9
DeiT-S⚗	22M	224^2	936.2	81.2	86.8	70.0
DeiT-B⚗	87M	224^2	290.9	83.4	88.3	73.2
DeiT-Ti⚗ / 1000 epochs	6M	224^2	2529.5	76.6	83.9	65.4
DeiT-S⚗ / 1000 epochs	22M	224^2	936.2	82.6	87.8	71.7
DeiT-B⚗ / 1000 epochs	87M	224^2	290.9	84.2	88.7	73.9
DeiT-B⚗ ↑384	87M	384^2	85.8	84.5	89.0	74.8
DeiT-B⚗ ↑384 / 1000 epochs	87M	384^2	85.8	85.2	89.3	75.2

* : *Regnet* optimized with a similar optimization procedure as ours, which boosts the results. These networks serve as teachers when we use our distillation strategy.

Table 5: Throughput on and accuracy on Imagenet [42], Imagenet Real [5] and Imagenet V2 matched frequency [41] of DeiT and of several state-of-the-art convnets, for models trained with no external data. The throughput is measured as the number of images that we can process per second on one 16GB V100 GPU. For each model we take the largest possible batch size for the usual resolution of the model and calculate the average time over 30 runs to process that batch.

2 DeiT (Data-efficient Image Transformers)

Hugo Touvron, et al. "Training data-efficient image transformers & distillation through attention," arXiv 2021

Training Details & Ablation

- Discuss **the DeiT training strategy** to **learn vision transformers in a data-efficient manner.**
- We build upon PyTorch [39] and the **timm** library [55]. (The timm implementation already included a training procedure that improved the accuracy of ViT-B from 77.91% to 79.35% top-1, and trained on Imagenet-1k with a 8xV100 GPU machine.)

❖ timm library
https://timm.fast.ai/
https://github.com/rwightman/pytorch-image-models

Initialization & hyper-parameters

- Transformers are relatively sensitive to initialization
- Follow Hanin et al. [20] to initialize the weights with a truncated normal distribution. Follow Cho et al. [9[] to select parameters τ=3.0, λ=0.1 for the usual (soft) distillation.

- We report the accuracy scores (%) after the initial training at resolution 224x224, and after fine-tuning at resolution 384x384. The hyper-parameters are fixed according to Table 9, and may be suboptimal.

Table 9: Ingredients and hyper-parameters for our method and Vit-B.

Methods	ViT-B [15]	DeiT-B
Epochs	300	300
Batch size	4096	1024
Optimizer	AdamW	AdamW
learning rate	0.003	$0.0005 \times \frac{\text{batchsize}}{512}$
Learning rate decay	cosine	cosine
Weight decay	0.3	0.05
Warmup epochs	3.4	5
Label smoothing ε	✗	0.1
Dropout	0.1	✗
Stoch. Depth	✗	0.1
Repeated Aug	✗	✓
Gradient Clip.	✓	✗
Rand Augment	✗	9/0.5
Mixup prob.	✗	0.8
Cutmix prob.	✗	1.0
Erasing prob.	✗	0.25

2 DeiT (Data-efficient Image Transformers)

Hugo Touvron, et al. "Training data-efficient image transformers & distillation through attention," arXiv 2021

Training Details & Ablation

Ablation study

Table 8: Ablation study on training methods on ImageNet [42]. The top row ("none") corresponds to our default configuration employed for DeiT. The symbols 3 and 7 indicates that we use and do not use the corresponding method, respectively. We report the accuracy scores (%) after the initial training at resolution 224x224, and after fine-tuning at resolution 384x384. The hyper-parameters are fixed according to Table 9, and may be suboptimal.

* indicates that the model did not train well, possibly because hyper-parameters are not adapted.

												top-1 accuracy	
Ablation on ↓	Pre-training	Fine-tuning	Rand-Augment	AutoAug	Mixup	CutMix	Erasing	Stoch. Depth	Repeated Aug.	Dropout	Exp. Moving Avg.	pre-trained 224^2	fine-tuned 384^2
none: DeiT-B	adamw	adamw	✓	✗	✓	✓	✓	✓	✓	✗	✗	81.8 ±0.2	83.1 ±0.1
optimizer	SGD	adamw	✓	✗	✓	✓	✓	✓	✓	✗	✗	74.5	77.3
	adamw	SGD	✓	✗	✓	✓	✓	✓	✓	✗	✗	81.8	83.1
data augmentation	adamw	adamw	✗	✗	✓	✓	✓	✓	✓	✗	✗	79.6	80.4
	adamw	adamw	✗	✓	✓	✓	✓	✓	✓	✗	✗	81.2	81.9
	adamw	adamw	✓	✗	✗	✓	✓	✓	✓	✗	✗	78.7	79.8
	adamw	adamw	✓	✗	✓	✗	✓	✓	✓	✗	✗	80.0	80.6
	adamw	adamw	✓	✗	✗	✗	✓	✓	✓	✗	✗	75.8	76.7
regularization	adamw	adamw	✓	✗	✓	✓	✗	✓	✓	✗	✗	4.3*	0.1
	adamw	adamw	✓	✗	✓	✓	✓	✗	✓	✗	✗	3.4*	0.1
	adamw	adamw	✓	✗	✓	✓	✓	✓	✗	✗	✗	76.5	77.4
	adamw	adamw	✓	✗	✓	✓	✓	✓	✓	✓	✗	81.3	83.1
	adamw	adamw	✓	✗	✓	✓	✓	✓	✓	✗	✓	81.9	83.1

2 DeiT Pytorch

gihub

facebookresearch/deit : https://github.com/facebookresearch/deit
lucidrains/vit-pytorch : https://github.com/lucidrains/vit-pytorch
FrancescoSaverioZuppichini/DeiT : https://github.com/FrancescoSaverioZuppichini/DeiT

timm : pytorch image models
rwightman/pytorch-image-models : https://github.com/rwightman/pytorch-image-models

FrancescoSaverioZuppichini/glasses : His deep learning computer vision library
https://github.com/FrancescoSaverioZuppichini/glasses

2 DeiT Pytorch

[참고] FrancescoSaverioZuppichini/DeiT :
https://github.com/FrancescoSaverioZuppichini/DeiT

- Knowledge distillation

```
import torch
from torch import nn
import torch.nn.functional as F
from torch import Tensor

class HardDistillationLoss(nn.Module):
    def __init__(self, teacher: nn.Module):
        super().__init__()
        self.teacher = teacher
        self.criterion = nn.CrossEntropyLoss()

    def forward(self, inputs: Tensor, outputs : Tensor, labels: Tensor) -> Tensor:

        base_loss = self.criterion(outputs, labels)

        with torch.no_grad():
            teacher_outputs = self.teacher(inputs)

        teacher_labels = torch.argmax(teacher_outputs, dim=1)
        teacher_loss = self.criterion(outputs, teacher_labels)

        return 0.5 * base_loss + 0.5 * teacher_loss

# little test
loss = HardDistillationLoss(nn.Linear(100, 10))
_ = loss(torch.rand((8, 100)), torch.rand((8, 10)), torch.ones(8).long())
```

→ Modify by Attention Distillation

```
from typing import Union

class HardDistillationLoss(nn.Module):
    def __init__(self, teacher: nn.Module):
        super().__init__()
        self.teacher = teacher
        self.criterion = nn.CrossEntropyLoss()

    def forward(self, inputs: Tensor, outputs: Union[Tensor, Tensor], labels: Tensor) -
> Tensor:
        # outputs contains booth predictions, one with the cls token and one with the
dist token
        outputs_cls, outputs_dist = outputs
        base_loss = self.criterion(outputs_cls, labels)

        with torch.no_grad():
            teacher_outputs = self.teacher(inputs)

        teacher_labels = torch.argmax(teacher_outputs, dim=1)
        teacher_loss = self.criterion(outputs_dist, teacher_labels)

        return 0.5 * base_loss + 0.5 * teacher_loss
```

2 DeiT Pytorch

[참고] FrancescoSaverioZuppichini/DeiT :
https://github.com/FrancescoSaverioZuppichini/DeiT

- Distillation token

```
from einops import rearrange, reduce, repeat
from einops.layers.torch import Rearrange, Reduce

class PatchEmbedding(nn.Module):
    def __init__(self, in_channels: int = 3, patch_size: int = 16, emb_size: int = 768,
img_size: int = 224):
        self.patch_size = patch_size
        super().__init__()
        self.projection = nn.Sequential(
            # using a conv layer instead of a linear one -> performance gains
            nn.Conv2d(in_channels, emb_size, kernel_size=patch_size,
stride=patch_size),
            Rearrange('b e (h) (w) -> b (h w) e'),
        )
        self.cls_token = nn.Parameter(torch.randn(1,1, emb_size))

        # distillation token
        self.dist_token = nn.Parameter(torch.randn(1,1, emb_size))

        self.positions = nn.Parameter(torch.randn((img_size // patch_size) **2 + 1,
emb_size))
```

```
    def forward(self, x: Tensor) -> Tensor:
        b, _, _, _ = x.shape
        x = self.projection(x)
        cls_tokens = repeat(self.cls_token, '() n e -> b n e', b=b)
        dist_tokens = repeat(self.dist_tokens, '() n e -> b n e', b=b)
        # prepend the cls token to the input
        x = torch.cat([cls_tokens, dist_tokens, x], dim=1)
        # add position embedding
        x += self.positions
        return x
```

2 DeiT Pytorch

- Classification Head

```
class ClassificationHead(nn.Module):
    def __init__(self, emb_size: int = 768, n_classes: int = 1000):
        super().__init__()

        self.head = nn.Linear(emb_size, n_classes)
        self.dist_head = nn.Linear(emb_size, n_classes)

    def forward(self, x: Tensor) -> Tensor:
        x, x_dist = x[:, 0], x[:, 1]
        x_head = self.head(x)
        x_dist_head = self.dist_head(x_dist)

        if self.training:
            x = x_head, x_dist_head
        else:
            x = (x_head + x_dist_head) / 2
        return x
```

[참고] FrancescoSaverioZuppichini/DeiT :
https://github.com/FrancescoSaverioZuppichini/DeiT

Follows the ViT code

```
class MultiHeadAttention(nn.Module):
    def __init__(self, emb_size: int = 768, num_heads: int = 8, dropout: float = 0):
        super().__init__()
        self.emb_size = emb_size
        self.num_heads = num_heads
        # fuse the queries, keys and values in one matrix
        self.qkv = nn.Linear(emb_size, emb_size * 3)
        self.att_drop = nn.Dropout(dropout)
        self.projection = nn.Linear(emb_size, emb_size)

    def forward(self, x : Tensor, mask: Tensor = None) -> Tensor:
        # split keys, queries and values in num_heads
        qkv = rearrange(self.qkv(x), "b n (h d qkv) -> (qkv) b h n d",
h=self.num_heads, qkv=3)
        queries, keys, values = qkv[0], qkv[1], qkv[2]
        # sum up over the last axis
        energy = torch.einsum('bhqd, bhkd -> bhqk', queries, keys) # batch,
num_heads, query_len, key_len
        if mask is not None:
            fill_value = torch.finfo(torch.float32).min
            energy.mask_fill(~mask, fill_value)

        scaling = self.emb_size ** (1/2)
        att = F.softmax(energy, dim=-1) / scaling
        att = self.att_drop(att)
        # sum up over the third axis
        out = torch.einsum('bhal, bhlv -> bhav ', att, values)
        out = rearrange(out, "b h n d -> b n (h d)")
        out = self.projection(out)
        return out
```

2 DeiT Pytorch

[참고] FrancescoSaverioZuppichini/DeiT :
https://github.com/FrancescoSaverioZuppichini/DeiT

Follows the ViT code

```
class ResidualAdd(nn.Module):
    def __init__(self, fn):
        super().__init__()
        self.fn = fn

    def forward(self, x, **kwargs):
        res = x
        x = self.fn(x, **kwargs)
        x += res
        return x

class FeedForwardBlock(nn.Sequential):
    def __init__(self, emb_size: int, expansion: int = 4, drop_p: float = 0.):
        super().__init__(
            nn.Linear(emb_size, expansion * emb_size),
            nn.GELU(),
            nn.Dropout(drop_p),
            nn.Linear(expansion * emb_size, emb_size),
        )
```

```
class TransformerEncoderBlock(nn.Sequential):
    def __init__(self,
                 emb_size: int = 768,
                 drop_p: float = 0.,
                 forward_expansion: int = 4,
                 forward_drop_p: float = 0.,
                 ** kwargs):
        super().__init__(
            ResidualAdd(nn.Sequential(
                nn.LayerNorm(emb_size),
                MultiHeadAttention(emb_size, **kwargs),
                nn.Dropout(drop_p)
            )),
            ResidualAdd(nn.Sequential(
                nn.LayerNorm(emb_size),
                FeedForwardBlock(
                    emb_size, expansion=forward_expansion, drop_p=forward_drop_p),
                nn.Dropout(drop_p)
            )
            ))

class TransformerEncoder(nn.Sequential):
    def __init__(self, depth: int = 12, **kwargs):
        super().__init__(*[TransformerEncoderBlock(**kwargs) for _ in range(depth)])
```

2 DeiT Pytorch

[참고] FrancescoSaverioZuppichini/DeiT :
https://github.com/FrancescoSaverioZuppichini/DeiT

DeiT model

```
class DeiT(nn.Sequential):
    def __init__(self,
                 in_channels: int = 3,
                 patch_size: int = 16,
                 emb_size: int = 768,
                 img_size: int = 224,
                 depth: int = 12,
                 n_classes: int = 1000,
                 **kwargs):
        super().__init__(
            PatchEmbedding(in_channels, patch_size, emb_size, img_size),
            TransformerEncoder(depth, emb_size=emb_size, **kwargs),
            ClassificationHead(emb_size, n_classes)
        )
```

To train, we can use a bigger model (ViT-Huge, RegNetY-16GF ...) as teacher and a smaller one (ViT-Small/Base) as student. The training code looks like this:

https://github.com/facebookresearch/deit

```
ds = ImageDataset('./imagenet/')
dl = DataLoader(ds, ...)

teacher = ViT.vit_large_patch16_224()
student = DeiT.deit_small_patch16_224()

optimizer = Adam(student.parameters())
criterion = HardDistillationLoss(teacher)

for data in dl:
    inputs, labels = data
    outputs = student(inputs)

    optimizer.zero_grad()

    loss = criterion(inputs, outputs, labels)

    loss.backward()
    optimizer.step()
```