AN IMAGE IS WORTH 16X16 WORDS: TRANSFORMERS FOR IMAGE RECOGNITION AT SCALE

💫 `ctrl+alt+t` 를 누르면 한 번에 여닫기를 할 수 있습니다.
그리고 화이트 모드로 보는 것을 권장드립니다.

CNN, Transformer를 전반적으로 짚어본 후, ViT에 대해서 자세하게 볼 예정입니다.

📢 **<Contents>**

0. 요약
    전반적으로 이런 내용을 다뤄요
1. CNN
    1-1. CNN Introduction
    1-2. CNN Application
    1-3. Max-pooling & Stride
    1-4. Design CNN Architecture
2. Transformer
    2-1. Introduction
    2-2. Multi-head Attention
    2-3. Encoder block
    2-4. Decoder with Masking
    2-5. Positional Encoding
    2-6. Learning rate warm-up and linear decay
    2-7. Appendix: Beyond the paper
3. 논문 리뷰
    3-1. Introduction
    3-2. Related Work
    3-3. Method
    3-4. Experiments
    3-5. Conclusion
4. Code
5. 추후 방향성
6. 참고 링크
7. ViT 이전에 참고할 만한 논문
8. ViT이후 관련 논문 for CLIP
9. 리뷰 후기

0. 요약

▼ 전반적으로 이런 내용을 다뤄요

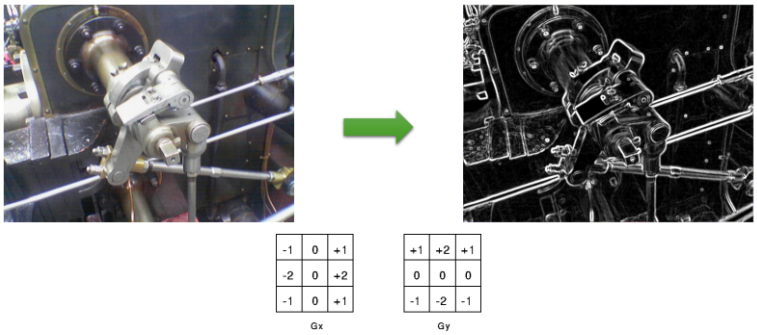
- ViT를 이해하기 전 CNN에 대해서 알아봅니다.
- 그리고 NLP분야에서 등장한 Transformer에 대해 알아봅니다.
- 이후, ViT에 대한 리뷰를 합니다.

1. CNN

▼ 1-1. CNN Introduction

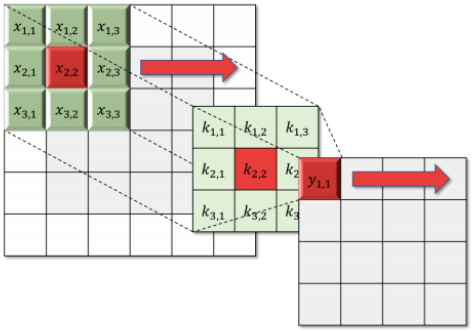
▼ Before DL

- Hand-crafted featrue(e.g. edge) 추출 후 모델

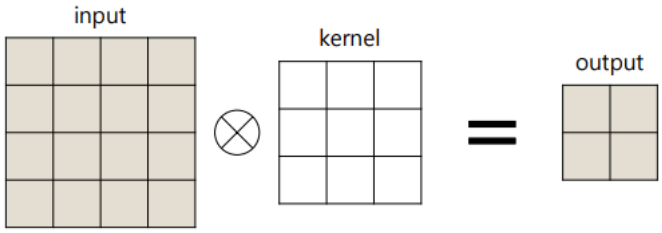


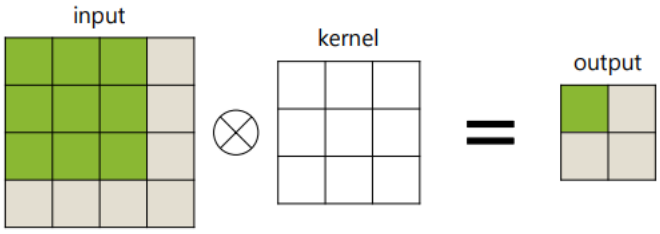
▼ Convolution Operation

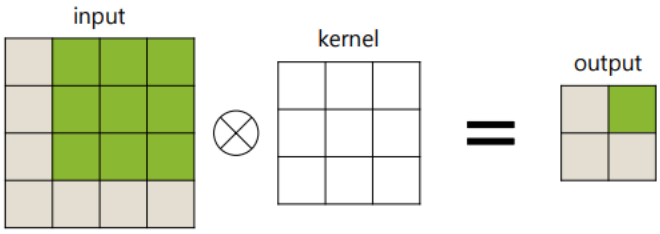
- Convolutional filter가 입력(x)에 대해서 돌아가며 동작

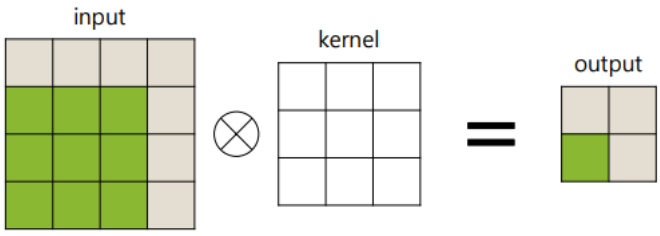

$x_{1,1}$ $x_{1,2}$ $x_{1,3}$
$x_{2,1}$ $x_{2,2}$ $x_{2,3}$
$x_{3,1}$ $x_{3,2}$ $x_{3,3}$
$k_{1,1}$ $k_{1,2}$ $k_{1,3}$
$k_{2,1}$ $k_{2,2}$
$k_{3,1}$ $k_{3,2}$
$y_{1,1}$

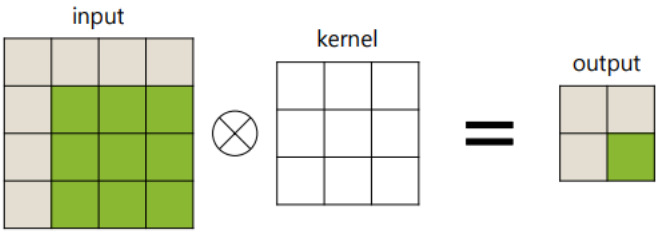
▼ CNN


input
kernel
output

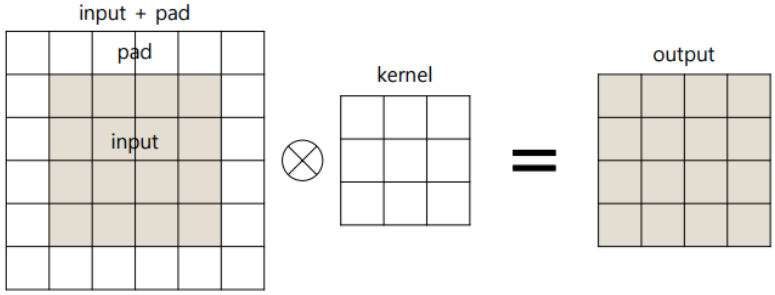

input
kernel
output

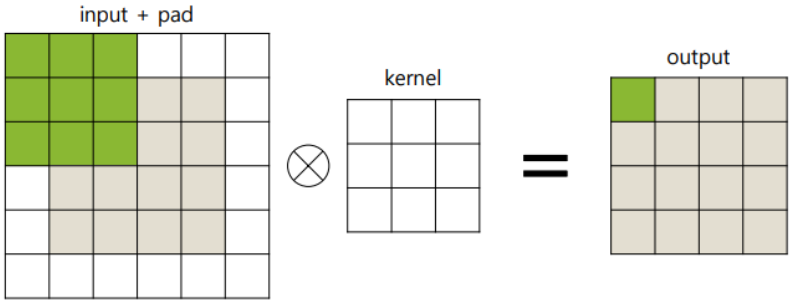

input
kernel
output


input
kernel
output



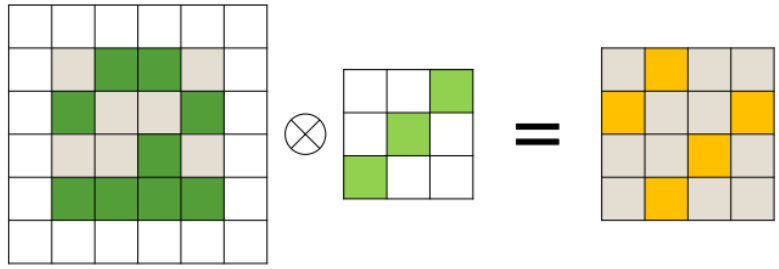
▼ CNN(with Pad)

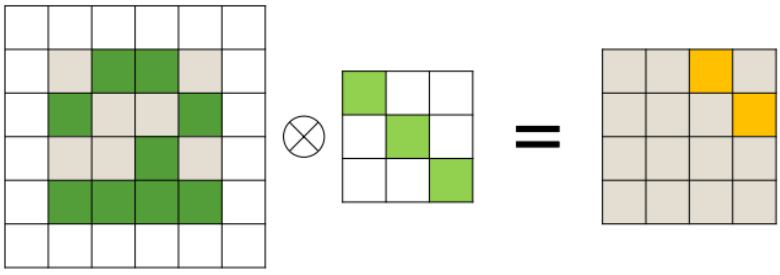


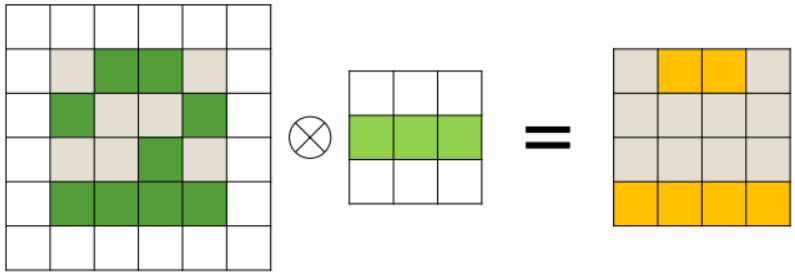


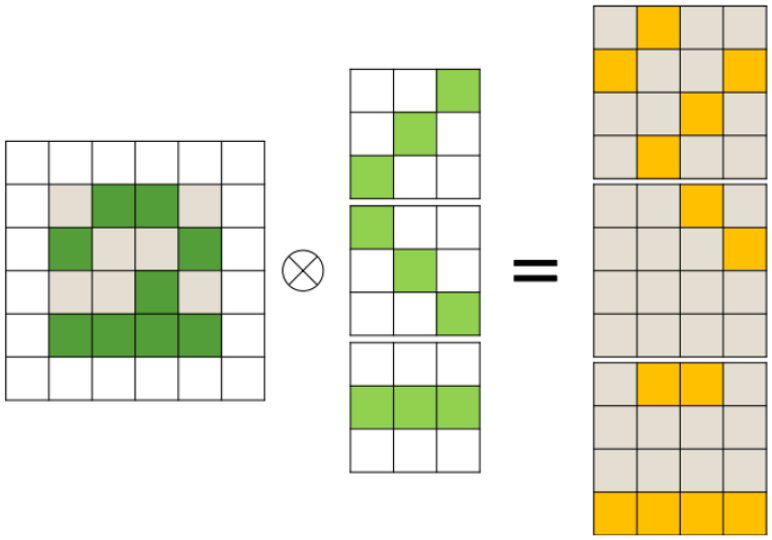
…

▼ How it works(intuitively)

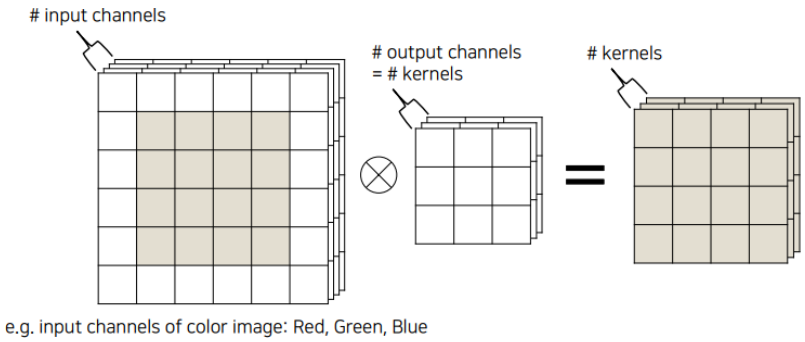








▼ Input&Output Channels



e.g. input channels of color image: Red, Green, Blue

▼ Input&Output Tensors

• Input Tensor Shape

• Output Tensor Shape

(N, C, H, W)

▼ 입출력 크기 계산 방법

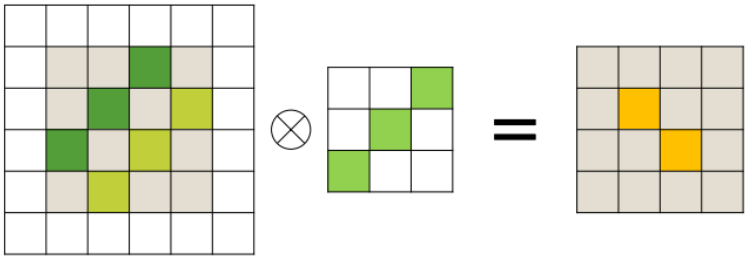
$$
\begin{aligned}
b &= \text{ batch size} \\
(x_{\text{height}}, x_{\text{width}}) &= \text{ input size} \\
c_{\text{in}} &= \#\text{ input channels} \\
c_{\text{out}} &= \#\text{ output channels} \\
(k_{\text{height}}, k_{\text{width}}) &= \text{ kernel size} \\
(y_{\text{height}}, y_{\text{width}}) &= \text{ output size} \\
y &= \text{conv 2 d}(x), \\
\text{where } &\begin{cases} |x| = (b, c_{\text{in}}, x_{\text{height}}, x_{\text{width}}) \\ |y| = (y_{\text{height}}, y_{\text{width}}) \\ \quad = (b, c_{\text{out}}, x_{\text{height}} - k_{\text{height}} + 1, x_{\text{width}} - k_{\text{width}} + 1). \end{cases}
\end{aligned}
$$

▼ 입출력 크기 계산 방법(+pad)

$$
\begin{aligned}
b &= \text{ batch size} \\
(x_{\text{height}}, x_{\text{width}}) &= \text{ input size} \\
c_{\text{in}} &= \#\text{ input channels} \\
c_{\text{out}} &= \#\text{ output channels} \\
(k_{\text{height}}, k_{\text{width}}) &= \text{ kernel size} \\
(y_{\text{height}}, y_{\text{width}}) &= \text{ output size} \\
(p_{\text{height}}, p_{\text{width}}) &= \text{ pad size} \\
y &= \text{conv 2 d}(x), \\
\text{where } &\begin{cases} |x| = (b, c_{\text{in}}, x_{\text{height}}, x_{\text{width}}) \\ |y| = (y_{\text{height}}, y_{\text{width}}) \\ \quad = (b, c_{\text{out}}, x_{\text{height}} + 2 \times p_{\text{height}} - k_{\text{height}} + 1, x_{\text{width}} + 2 \times p_{\text{wi}} \end{cases}
\end{aligned}
$$

▼ Convolution Layer의 특징

- Feature의 위치에 구애받지 않는다.

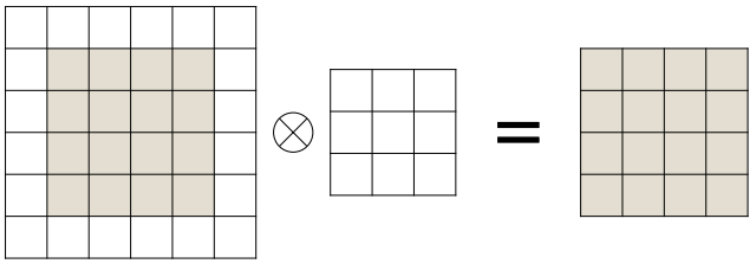


▼ Convolution Layer의 장점

- 같은 입출력을 갖는 FC Layer에 비해 더 작은 weight를 가진다.

$$
\begin{aligned}
y &= \text{conv 2 d}(x), \\
\text{where } &\begin{cases} |x| = (b, c_{\text{in}}, x_{\text{height}}, x_{\text{width}}) \\ |y| = (y_{\text{height}}, y_{\text{width}}) \\ \quad = (b, c_{\text{out}}, x_{\text{height}} - k_{\text{height}} + 1, x_{\text{width}} - k_{\text{width}} + 1). \end{cases}
\end{aligned}
$$

- 병렬 계산 구성이 쉬우므로, GPU에서의 연산이 매우 빠르다.



▼ Convolution Layer의 단점

- FC Layer에 비해 입출력 크기 계산이 까다로워, 네트워크 구성이 쉽지 않다.

$$y = \mathrm{conv2d}(x),$$

$$\text{where} \begin{cases} |x| = (b, c_{\text{in}}, x_{\text{height}}, x_{\text{width}}) \\ |y| = (y_{\text{height}}, y_{\text{width}}) \\ \quad = (b, c_{\text{out}}, x_{\text{height}} + 2 \times p_{\text{height}} - k_{\text{height}} + 1, x_{\text{width}} + 2 \times \end{cases}$$

▼ Summary

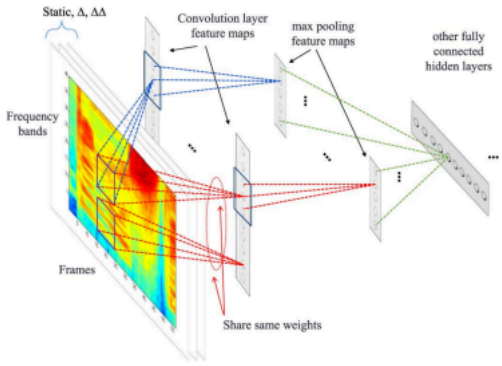
- 예전부터 영상 처리 분야에서는 Convolution Filter를 활용
  - 탐지하고자 하는 패턴에 따라 필터를 손으로 하나하나 깎아 구성
- CNN은 데이터셋의 x와 y의 구성에 따라 자동으로 탐지해야 할 패턴을 추출
  - 따라서 내부 kernel은 해당 패턴을 추출하기 위한 형태로 자동으로 구성됨
- FC Layer에 비해 매우 빠르고 적은 weight parameter를 가짐

▼ 1-2. CNN Application

- Computer Vision

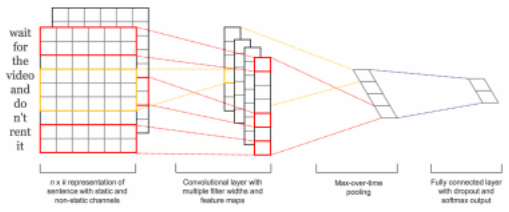
[Mirzaei et al., 2014]

- Speech Recognition



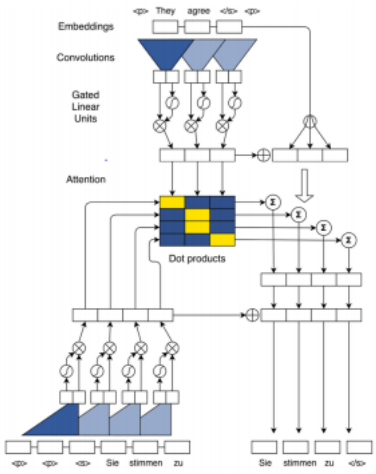
[Abdel-Hamid et al., 2014]

- Text Classification



[Kim, 2014]

- Machine Translation



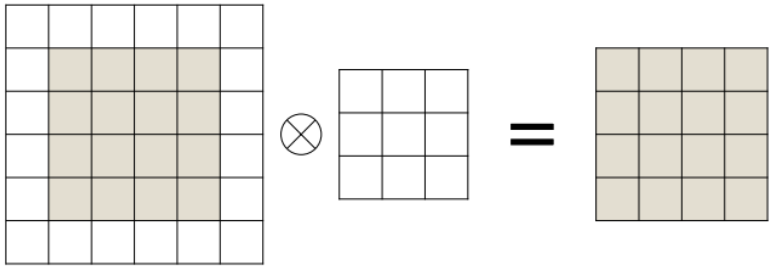
[Gehring et al., 2014]

- Time Series

▼ 1-3. Max-pooling & Stride

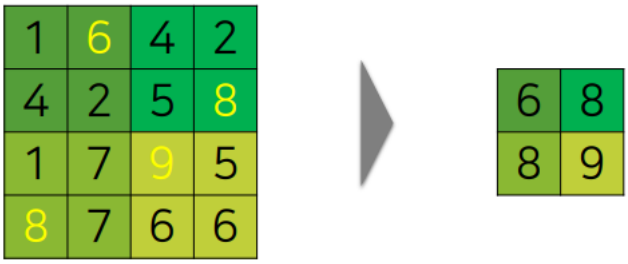
▼ Dimenstion Reduction

- 고차원 공간의 sparse한 데이터를 저차원 공간에 mapping
  - 그 과정에서 복잡하게 얽혀 있는(entangle) 데이터를 풀어냄
- 하지만, '3x3 + 1 padding' conv layer는 입출력 텐서의 크기가 같음



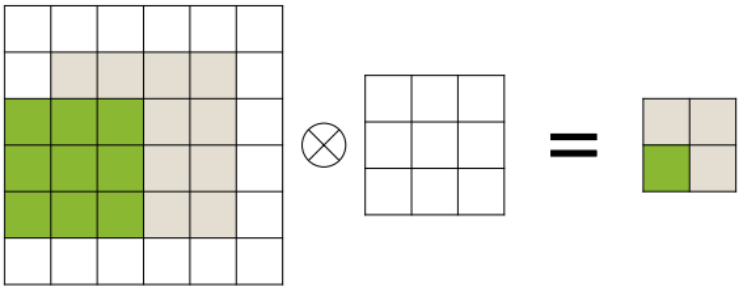
▼ Max-pooling

- Down sampling 방법



▼ Stride

- Working in convolution layer



▼ Dimension Reductions in CNN

- Max-pooling
  - 별도의 max-pooling layer를 활용
  - 초기에 많이 됨
- Stride
  - 같은 conv layer 내에서 간단히 동작
  - 근래에 좀 더 애용되는 추세

$(p_{\text{height}}, p_{\text{width}})$ = pad size
$(s_{\text{height}}, s_{\text{width}})$ = stride size
$y = \text{conv 2 d}(x)$,
where

$$\begin{cases} |x| = (b, c_{\text{in}}, x_{\text{height}}, x_{\text{width}}) \\ |y| = (y_{\text{height}}, y_{\text{width}}) \\ \quad = \left(b, c_{\text{out}}, \left\lfloor \frac{x_{\text{height}} + 2 \times p_{\text{height}} - (k_{\text{height}} - 1) - 1}{s_{\text{height}}} + 1 \right\rfloor, \left| \frac{x_{\text{width}}} \right. \right. \end{cases}$$

▼ 1-4. Design CNN Architecture

▼ CNN의 특징

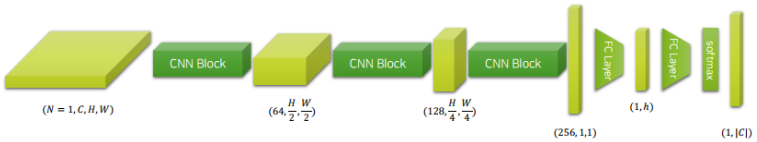
- FC Layer에 비해 입출력 크기가 계산이 까다로워, 네트워크 구성이 쉽지 않다.

$$y = \mathrm{conv}\,2\,\mathrm{d}(x),$$

$$\text{where} \begin{cases} |x| = (b, c_{\text{in}}, x_{\text{height}}, x_{\text{width}}) \\ |y| = (y_{\text{height}}, y_{\text{width}}) \\ \quad = \left(b, c_{\text{out}}, \left[\dfrac{x_{\text{height}} + 2 \times p_{\text{height}} - (k_{\text{height}} - 1) - 1}{s_{\text{height}}} + 1\right]\right. \end{cases}$$

▼ Tip

- CNN Block



1. 3×3 Convolution Layer
2. ReLU
3. Batch Normalization
4. 3×3 Convolution Layer (+ with Stride size (2×2))
5. ReLU
6. Batch Normalization
7. (+ Max-pooling if no stride)

https://github.com/dorae222/DeepLearning/tree/main/2.%20Lv2_Beginning_DL/06-cnn

---

2. Transformer

▼ 2-1. Introduction

▼ SInce 2016,

- In 2016, Google puplished thier nueral machine translation system(GNMT), which outperforms previous traditional MT system.

Google's Neural Machine Translation System.pdf

Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation

Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V. Le, Mohammad Norouzi
yonghui,schuster,zhifengc,qvl,mnorouzi@google.com
Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, Jeff Klingner, Apurva Shah, Melvin Johnson, Xiaobing Liu, Łukasz Kaiser, Stephan Gouws, Yoshikiyo Kato, Taku Kudo, Hideto Kazawa, Keith Stevens, George Kurian, Nishant Patil, Wei Wang, Cliff Young, Jason Smith, Jason Riesa, Alex Rudnick, Oriol Vinyals, Greg Corrado, Macduff Hughes, Jeffrey Dean

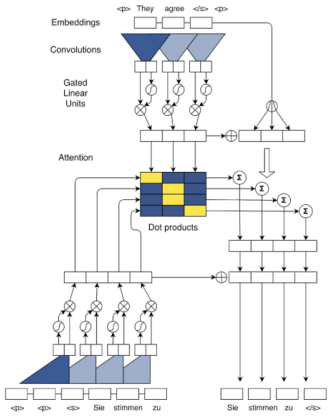
- However, RNN based Sequence-to-sequence reveals its limitations.

Table 10: Mean of side-by-side scores on production data

	PBMT	GNMT	Human	Relative Improvement
English → Spanish	4.885	5.428	5.504	87%
English → French	4.932	5.295	5.496	64%
English → Chinese	4.035	4.594	4.987	58%
Spanish → English	4.872	5.187	5.372	63%
French → English	5.046	5.343	5.404	83%
Chinese → English	3.694	4.263	4.636	60%

▼ Fully Convilutional Seq2Seq[Gehring et al.2017p

Convolutional Sequence to Sequence Learning.pdf



WMT'16 English-Romanian	BLEU
Sennrich et al. (2016b) GRU (BPE 90K)	28.1
ConvS2S (Word 80K)	29.45
ConvS2S (BPE 40K)	30.02

WMT'14 English-German	BLEU
Luong et al. (2015) LSTM (Word 50K)	20.9
Kalchbrenner et al. (2016) ByteNet (Char)	23.75
Wu et al. (2016) GNMT (Word 80K)	23.12
Wu et al. (2016) GNMT (Word pieces)	24.61
ConvS2S (BPE 40K)	25.16

WMT'14 English-French	BLEU
Wu et al. (2016) GNMT (Word 80K)	37.90
Wu et al. (2016) GNMT (Word pieces)	38.95
Wu et al. (2016) GNMT (Word pieces) + RL	39.92
ConvS2S (BPE 40K)	40.51

Table 1. Accuracy on WMT tasks comapred to previous work. ConvS2S and GNMT results are averaged over several runs.

▼ Attention is all you need

Attention Is All You Need.pdf

▼ Transformer

- Encoder + decoder

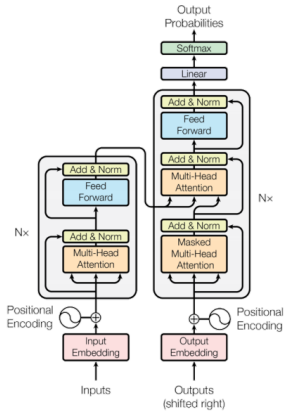


Figure 1: The Transformer - model architecture.

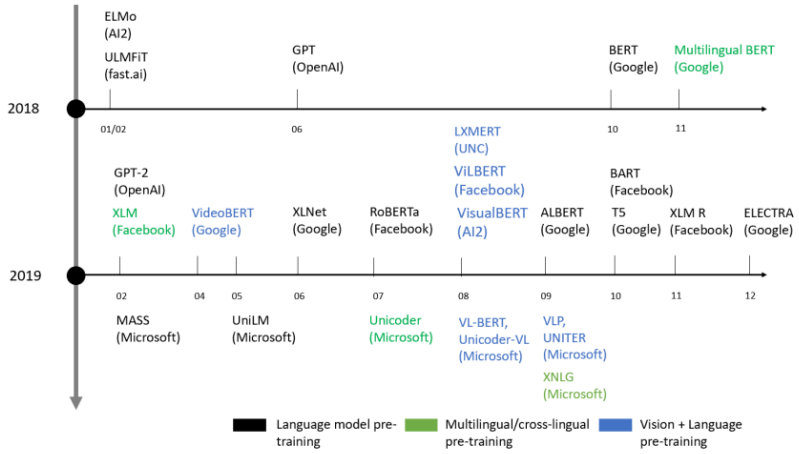
- 성능과 속도 모두 기존 모델을 압도

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

Model	BLEU		Training Cost (FLOPs)	
	EN-DE	EN-FR	EN-DE	EN-FR
ByteNet [18]	23.75			
Deep-Att + PosUnk [39]		39.2		$1.0 \cdot 10^{20}$
GNMT + RL [38]	24.6	39.92	$2.3 \cdot 10^{19}$	$1.4 \cdot 10^{20}$
ConvS2S [9]	25.16	40.46	$9.6 \cdot 10^{18}$	$1.5 \cdot 10^{20}$
MoE [32]	26.03	40.56	$2.0 \cdot 10^{19}$	$1.2 \cdot 10^{20}$
Deep-Att + PosUnk Ensemble [39]		40.4		$8.0 \cdot 10^{20}$
GNMT + RL Ensemble [38]	26.30	41.16	$1.8 \cdot 10^{20}$	$1.1 \cdot 10^{21}$
ConvS2S Ensemble [9]	26.36	**41.29**	$7.7 \cdot 10^{19}$	$1.2 \cdot 10^{21}$
Transformer (base model)	27.3	38.1	**$3.3 \cdot 10^{18}$**	
Transformer (big)	**28.4**	**41.8**	$2.3 \cdot 10^{19}$	

▼ Transformer and MLP

- Pretraining and finetuning (Transfer Learning) with Big-LM.



▼ 2-2. Multi-head Attention

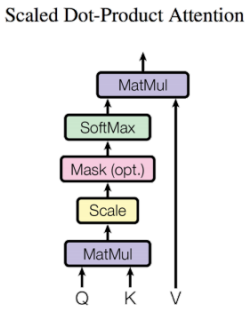
▼ Attention: Query Generation

- Example

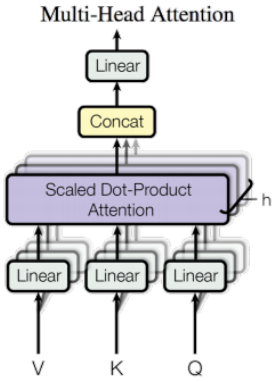
- 마음의 상태(state)를 잘 반영하면서 좋은 검색 결과를 이끌어내는 쿼리를 얻기 위함
- 만약 검색을 다양하게 할 수 있다면?

▼ Transformer & Attention

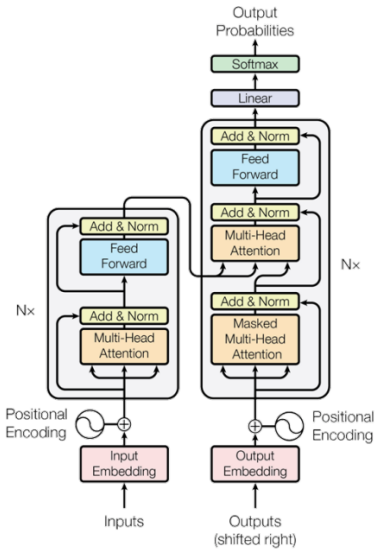
▼ Scaled Dot-product Attention



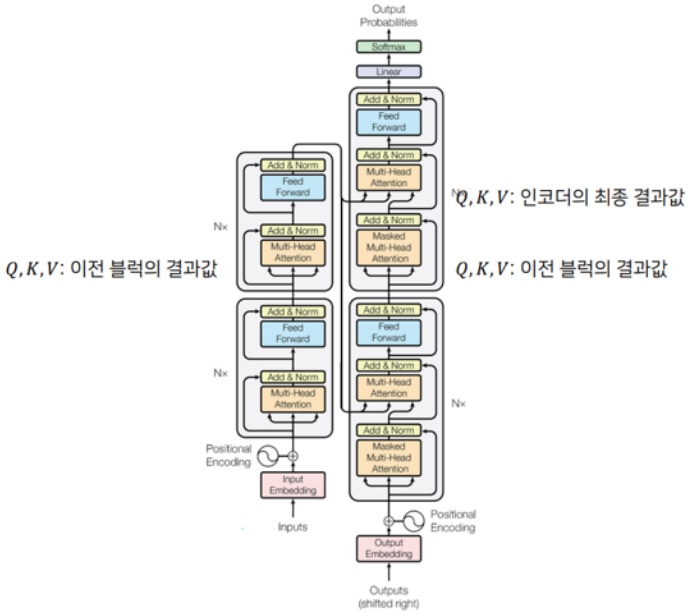
▼ Multi-Head Attention



▼ Transformer



▼ Encoder&Decoder가 여러 층인 것을 굳이 그려보자면



▼ Equations

Multihead_Attention.pdf

In case of Q, K and V come from same origin,
$|Q| = |K| = |V| = (\text{batch\_size}, n \text{ or } m, \text{hidden\_size})$.

In case of Q and K, V come from different origin,
$|Q| = (\text{batch\_size}, n, \text{hidden\_size})$
$|K| = |V| = (\text{batch\_size}, m, \text{hidden\_size})$.

$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{Q \cdot K^\intercal}{\sqrt{d_{\text{head}}}}) \cdot V$$

$$\text{MultiHead}(Q, K, V) = [head_1; \cdots; head_h] \cdot W^O$$

$$\text{where } head_i = \text{Attention}(Q \cdot W_i^Q, K \cdot W_i^K, V \cdot W_i^V),$$

$$\text{and } W_i^Q \in \mathbb{R}^{d_{\text{model}} \times d_{\text{head}}}, W_i^K \in \mathbb{R}^{d_{\text{model}} \times d_{\text{head}}}, W_i^V \in \mathbb{R}^{d_{\text{model}} \times d_{\text{head}}}, W^O \in \mathbb{R}^{(h \times d_{\text{head}}) \times d_{\text{model}}}.$$

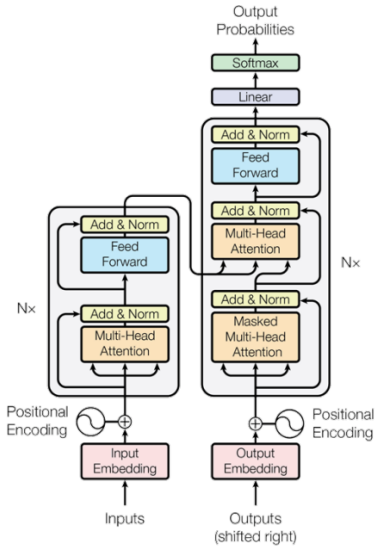
$$d_{\text{head}} = d_{\text{model}}/h = 64$$
$$h = 8, d_{\text{model}} = 512$$

▼ Summary

- Previous method: attention in sequence to sequence
  - Query를 잘 만들어 key-value를 잘 matching시키자
- Multi-head Attention
  - 여러개의 Query를 만들어 다양한 정보를 잘 얻어오자
- Attention 자체로도 정보의 encoding과 decoding이 가능함을 보여줌

▼ 2-3. Encoder block

▼ Transformer



▼ Equations

▼ Q,K and V are from previous alayer:

- Residual connections and Layer Normalizations are used.

Deep Residual Learning for Image Recognition.pdf

Layer Normalization.pdf

$$h_{0,1:m} = \text{emb}(x_{1:m}) + \text{pos}(1, m)$$
$$\tilde{h}^{\text{enc}}_{i,1:m} = \text{LayerNorm}(\text{Multihead}_i(Q, K, V) + h^{\text{enc}}_{i-1,1:m}),$$
$$\text{where } Q = K = V = h^{\text{enc}}_{i-1,1:m}.$$

$$\text{FFN}(h_{i,t}) = \text{ReLU}(h_{i,t} \cdot W^1_i) \cdot W^2_i$$
$$\text{where } W^1_i \in \mathbb{R}^{d_{\text{model}} \times d_{\text{ff}}} \text{ and } W^2_i \in \mathbb{R}^{d_{\text{ff}} \times d_{\text{model}}}.$$
$$h^{\text{enc}}_{i,1:m} = \text{LayerNorm}([\text{FFN}(\tilde{h}^{\text{enc}}_{i,1}); \cdots; \text{FFN}(\tilde{h}^{\text{enc}}_{i,m})] + \tilde{h}_{i,1:m})$$

▼ Encoder is stack of encoder blocks:

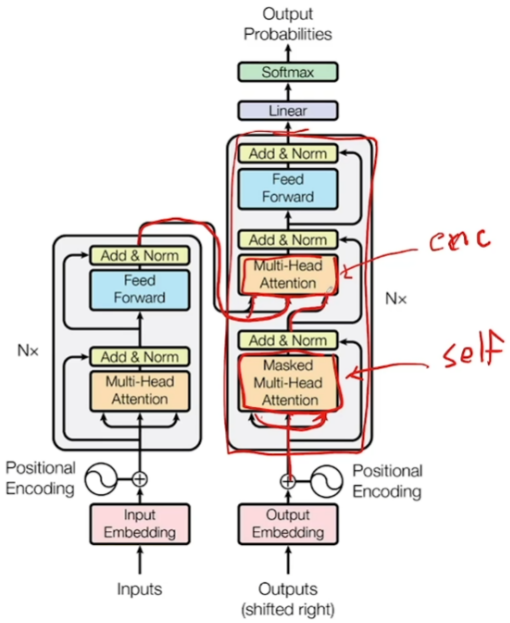
$$h^{\text{enc}}_{\ell_{\text{enc}},1:m} = \text{Block}_{\text{enc}}(h^{\text{enc}}_{\ell_{\text{enc}}-1,1:m})$$
$$\cdots$$
$$h^{\text{enc}}_{1,1:m} = \text{Block}_{\text{enc}}(h^{\text{enc}}_{0,1:m})$$

▼ Summary

- Encoder는 self-attention으로 구성되어 있음
  - Q,K,V는 이전 레이어의 출력값 - 즉, 같은 값
- Seq2Seq의 Attention과 달리, Q도 모든 time-step을 동시에 연산
  - 빠르지만 메모리를 많이 먹게 됨
- Residual connection으로 인해 깊은 네트워크 구성 가능
  - Big LM의 토대 마련

▼ 2-4. Decoder with Masking

▼ Transformer



▼ Equations

▼ Given Dataset.

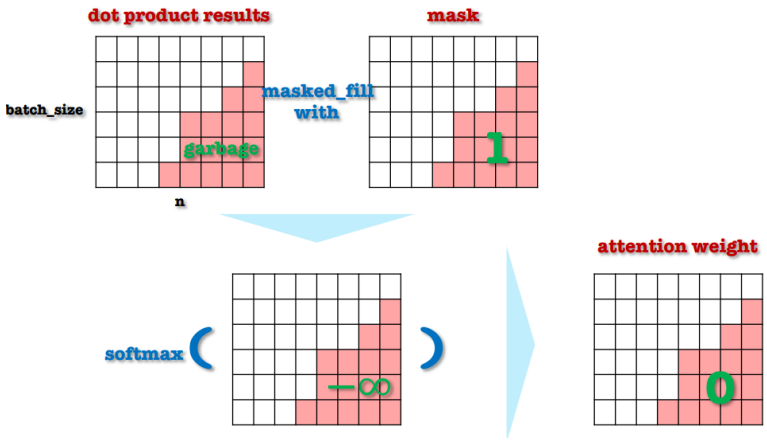
$$\mathcal{D} = \{x^i, y^i\}_{i=1}^N$$
$$x^i = \{x_1^i, \cdots, x_m^i\} \text{ and } y^i = \{y_0^i, y_1^i, \cdots, y_n^i\},$$
$$\text{where } y_0 = <\text{BOS}> \text{ and } y_n = <\text{EOS}>.$$

▼ What we want is

$$\hat{y}_{1:n} = f(x_{1:m} : \theta)$$

▼ Before we start

- Using mask, assign -∞ to make 0s for softmax results..



▼ Decoder Self-attention with mask

- 모든 attention에는 <pad>에 마스킹이 들어간다.
- 단, 디코더에서는 AuroRegressive한 특성으로 인해, 다음스텝을 보는 것을 방지하는 마스킹을 함께 해줘야 한다.

$$h_{0,1:n} = \text{emb}(y_{0:n-1}) + \text{pos}(0, n-1)$$
$$\tilde{h}^{\text{dec}}_{i,1:n} = \text{LayerNorm}(\text{Multihead}_i(Q,K,V) + h^{\text{dec}}_{i-1,1:n}),$$
$$\text{where } Q = K = V = h^{\text{dec}}_{i-1,1:n}.$$

▼ Attention from encoder with mask for <pad>

$$\tilde{h}^{\text{dec}}_{i,1:n} = \text{LayerNorm}(\text{Multihead}_i(Q,K,V) + h^{\text{dec}}_{i-1,1:n}),$$
$$\text{where } Q = \tilde{h}^{\text{dec}}_{i,1:n} \text{ and } K = V = h^{\text{dec}}_{\ell,1:m}.$$

▼ FC layers

$$\text{FFN}(h_{i,t}) = \text{ReLU}(h_{i,t} \cdot W^1_i) \cdot W^2_i$$
$$\text{where } W^1_i \in \mathbb{R}^{d_{\text{model}} \times d_{\text{ff}}} \text{ and } W^2_i \in \mathbb{R}^{d_{\text{ff}} \times d_{\text{model}}}.$$

$$h^{\text{dec}}_{i,1:m} = \text{LayerNorm}([\text{FFN}(\tilde{h}^{\text{dec}}_{i,1}); \cdots; \text{FFN}(\tilde{h}^{\text{dec}}_{i,m})] + \tilde{h}^{\text{dec}}_{i,1:m})$$

▼ Decoder is stack of decoder blocks.

$$h^{\text{dec}}_{\ell_{\text{dec}},1:m} = \text{Block}_{\text{dec}}(h^{\text{dec}}_{\ell_{\text{dec}}-1,1:m})$$
$$\cdots$$
$$h^{\text{dec}}_{1,1:m} = \text{Block}_{\text{dec}}(h^{\text{dec}}_{0,1:m})$$

▼ Generator

$$\hat{y}_{1:n} = \text{softmax}(h^{\text{dec}}_{\ell_{\text{dec}},1:m} \cdot W_{\text{gen}}),$$
$$\text{where } h^{\text{dec}}_{\ell_{\text{dec}},1:m} \in \mathbb{R}^{\text{batch\_size} \times n \times \text{hidden\_size}} \text{ and } W_{\text{gen}} \in \mathbb{R}^{\text{hidden\_size} \times |V|}.$$

▼ Summary

- Decoder는 2가지의 Attention으로 구성됨
  - Attention from encoder:
    - K와 V는 encoder의 최종 출력 값, Q는 이전 레이어의 출력 값
  - Self-Attention with mask:
    - Q,K,V는 이전 레이어의 출력 값
    - Attention weight 계산 시, softmax 연산 이전에 masking을 통해 음의 무한대를 주어, 미래 time-step을 보는 것을 방지
- 추론 때에는 self-attention의 mask는 필요 없으나, 모든 layer의 t 시점 이전의 모든 time-step(<t)의 hidden_state가 필요

▼ 2-5. Positional Encoding

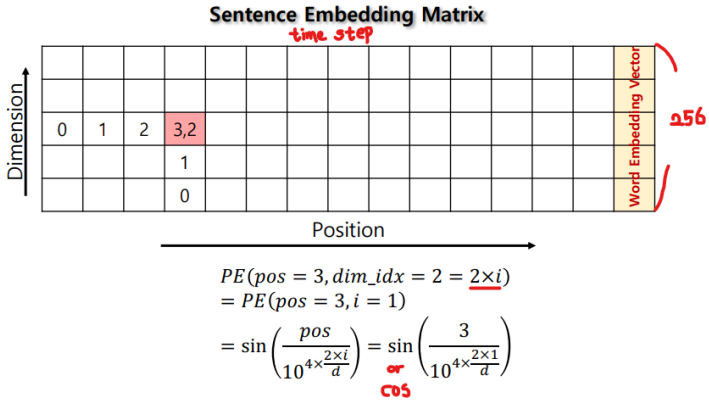
▼ Unlike RNN,

- Transformer는 위치 정보를 스스로 처리하지 않음(Conv2S도 마찬가지)
  - $h_t = f(x_t, h_{t-1})$ → RNN 계열은 위치 정보를 스스로 처리했음
  - 마치 FC layer의 입력 feature 순서를 바꿔 학습해도 성능이 똑같은 것과 같음

- 입력 순서를 바꿔 넣으면 출력도 순서가 바뀐 채 같은 값이 나올 것
- 따라서 위치순서 정보를 따로 인코딩해서 넣어줘야 함

▼ Positonal Encoding

- 기존의 word embedding 값에 positonal encoding 값을 더해줌



▼ vs Positional Embedding → 학습도 가능

- 사실 위치 정보도 integer 값이므로 embedding layer를 통해 임베딩 할 수 있음
- BERT와 같은 모델은 positional encoding 대신에 positional embedding을 사용하기도함

▼ Summary

- RNN과 달리, 순서(위치) 정보를 encoding해주는 작업이 필요
  - 학습이 아닌 단순 계산 후 encoding
- 학습에 의해 달라지는 값이 아니므로, 한번만 계산해 놓으면 됨

https://www.blossominkyung.com/deeplearning/transfomer-positional-encoding

▼ 2-6. Learning rate warm-up and linear decay

▼ Previous Method

SGD+Gradient Clipping

- 가장 기본적인 방법

$\theta \leftarrow \theta - \gamma \bigtriangledown_\theta L(\theta)$

- Learning rate에 따른 성능 변화
- 학습 후반부에 LR decay 해주기도

Adam

- Adaptive하게 LR을 조절

Algorithm 1: Generic adaptive optimization method setup. All operations are element-wise.

Input: $\{\alpha_t\}_{t=1}^T$: step size, $\{\phi_t, \psi_t\}_{t=1}^T$: function to calculate momentum and adaptive rate, θ_0: initial parameter, $f(\theta)$: stochastic objective function.
Output: θ_T: resulting parameters
while $t = 1$ *to* T **do**
  $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Calculate gradients w.r.t. stochastic objective at timestep t)
  $m_t \leftarrow \phi_t(g_1, \cdots, g_t)$ (Calculate momentum)
  $l_t \leftarrow \psi_t(g_1, \cdots, g_t)$ (Calculate adaptive learning rate)
  $\theta_t \leftarrow \theta_{t-1} - \alpha_t m_t l_t$ (Update parameters)
return θ_T

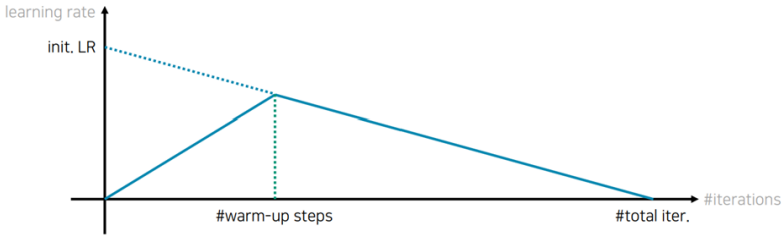
출처: [Liu et al., 2020]

- 일부 깊은 네트워크(e.g.Transformer)에서 서 성능이 낮음
  - 문제는 지금은 Transformer의 세상

▼ Warm-up and Linear Decay(Noam Decay)

- Heuristic Methods
  - Control learning rate for Adam with hyper-params

- 학습 초기 불안정한 gradient를 통해 잘못된 momentum을 갖는 것을 방지
  - Residual Connection을 하는 과정에서 발생??
  - 대체로 5% 근처



- 결국 Trial&Error방식으로 Hyper-parameter 튜닝을 해야함
  - 가장 핵심은 #warm-up steps와 #total iterations.
  - 이외에도 다양한 hyper-parmas: init LR, batch size
- 심지어 튜닝에 따라 SGD+Gradient Clipping이 더 나은 결과를 얻기도 함

▼ Rectified Adam[Liu et al., 2020]

> On The Variance of the Adaptive Learning Rate and Beyond.pdf

- Adam이 잘 동작하지 않는 이유(가설)
- Due to the lack of samples in the early stage, the adaptive learning rate has an undesirably large variance, which leads to suspicious/bad local optima. – [Liu et al., 2020]

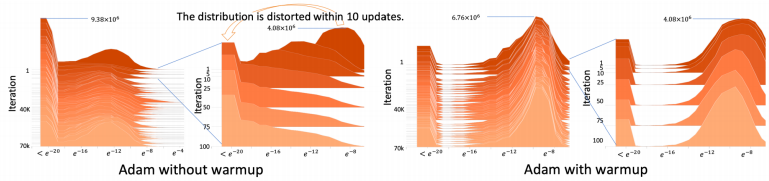


Figure 2: The absolute gradient histogram of the Transformers on the De-En IWSLT' 14 dataset during the training (stacked along the y-axis). X-axis is absolute value in the log scale and the height is the frequency. Without warmup, the gradient distribution is distorted in the first 10 steps.

- Pytorch 구현

> https://github.com/LiyuanLucasLiu/RAdam

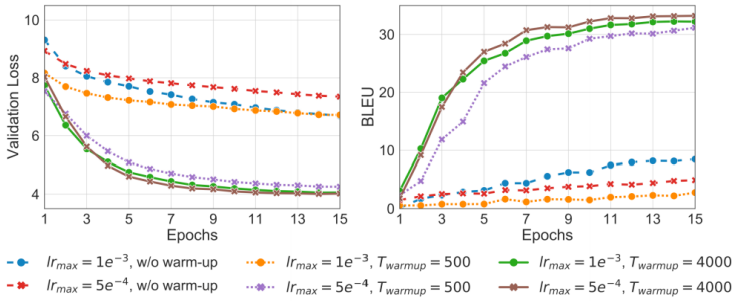
  - `$pip install torch-optimizer`

▼ 2-7. Appendix: Beyond the paper

▼ Transformer의 단점

▼ 학습이 까다롭다.

- Bad local optima에 빠지기 매우 쉬움
- 그런데 paper에서 이것을 언급하지 않음
  - #warm-up step, learning rate



(a) Loss/BLEU on the IWSLT14 De-En task (Adam)

▼ 오죽하면, 아래와 같은 추가 논문들이 제안됨

Training Tips for the Transformer Model.pdf

Transformers without Tears_Improving the Normalization of Self-Attention.pdf

On the Variance of the Adaptive Learning Rate and Beyond.pdf

▼ On Layer Normalization in Transformer Architecture[Xiong et al., 2020]

On Layer Normalization in Transformer Architecture.pdf

▼ Previous Work:

- Use Noam decay(warm-up and linear decay)
- Rectified Adam(RAdam)

▼ Propose:

- Layer Norm의 위치에 따라 학습이 수월해짐
  - LN이 gradient를 평탄하게 바꾸는 효과

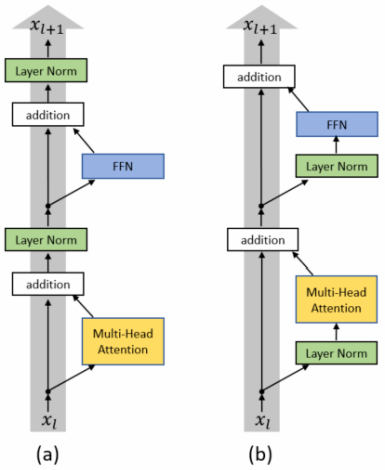
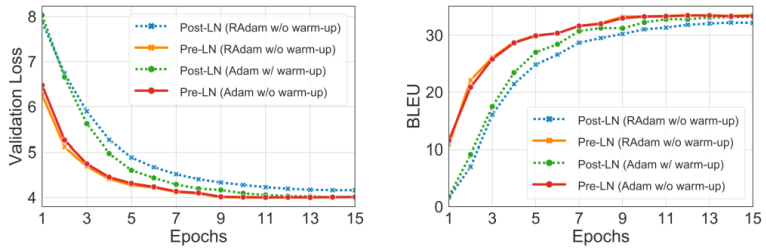


Figure 1: (a) Post-LN Transformer layer; (b) Pre-LN Transformer layer.

Table 1: Post-LN Transformer v.s. Pre-LN Transformer

Post-LN Transformer	Pre-LN Transformer
$x_{l,i}^{post,1} = \text{MultiHeadAtt}(x_{l,i}^{post}, [x_{l,1}^{post}, \cdots, x_{l,n}^{post}])$	$x_{l,i}^{pre,1} = \text{LayerNorm}(x_{l,i}^{pre})$
$x_{l,i}^{post,2} = x_{l,i}^{post} + x_{l,i}^{post,1}$	$x_{l,i}^{pre,2} = \text{MultiHeadAtt}(x_{l,i}^{pre,1}, [x_{l,1}^{pre,1}, \cdots, x_{l,n}^{pre,1}])$
$x_{l,i}^{post,3} = \text{LayerNorm}(x_{l,i}^{post,2})$	$x_{l,i}^{pre,3} = x_{l,i}^{pre} + x_{l,i}^{pre,2}$
$x_{l,i}^{post,4} = \text{ReLU}(x_{l,i}^{post,3} W^{1,l} + b^{1,l}) W^{2,l} + b^{2,l}$	$x_{l,i}^{pre,4} = \text{LayerNorm}(x_{l,i}^{pre,3})$
$x_{l,i}^{post,5} = x_{l,i}^{post,3} + x_{l,i}^{post,4}$	$x_{l,i}^{pre,5} = \text{ReLU}(x_{l,i}^{pre,4} W^{1,l} + b^{1,l}) W^{2,l} + b^{2,l}$
$x_{l+1,i}^{post} = \text{LayerNorm}(x_{l,i}^{post,5})$	$x_{l+1,i}^{pre} = x_{l,i}^{pre,5} + x_{l,i}^{pre,3}$
	Final LayerNorm: $x_{Final,i}^{pre} \leftarrow \text{LayerNorm}(x_{L+1,i}^{pre})$

▼ Evaluation Results



(a) Validation Loss (IWSLT) (b) BLEU (IWSLT)

▼ Summary

- Pre-Norm 방식을 통해 warm-up 및 LR 튜닝 제거 가능
  - LR decay는 여전히 필요
- 그 밖에도 Layer Norm을 대체하거나, weight initialization을 활용하여 좀 더 나은 성능을 확보할 수 있음

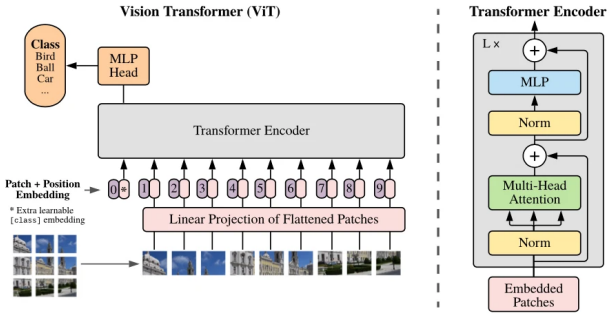
---

3. 논문 리뷰

💡 앞의 1~2까지 내용을 보시고 아래 내용을 보시면 아래 논문을 이해하기 쉬울 것 같습니다.

https://arxiv.org/pdf/2010.11929.pdf

▼ 3-1. Introduction



▼ 논문은 CNN(Convolutional Neural Networks)을 대체할 수 있는 새로운 아키텍처로 Transformer를 제시합니다.

- 기존 NLP분야에서 우수한 성능을 보이고 있는 Transformer를 CNN 대신 사용하고자 합니다.

▼ CNN은 지역적인 정보를 중점적으로 다루지만, Transformer는 전역적인 정보도 쉽게 활용할 수 있다는 장점이 있습니다.

- Inductive Bias 해소 → 기존 CNN의 문제점
  - Translation Equivariance & Locality

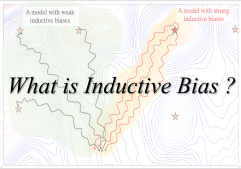
▼ Inductive Bias란

- 정리하기에 시간이 부족해 정리가 잘 된 링크 첨부합니다.

[머신러닝/딥러닝] Inductive Bias란?

Inductive Bias란 무엇일까요? 최근 논문들을 보면 그냥 Bias도 아니고 inductive Bias라는 말이 자주 나오는 것을 확인할 수 있는데요! 오늘은 해당 개념에 대해 정리해보는 시간을 가지려고 합니

https://velog.io/@euisuk-chung/Inductive-Bias란



- 이미지를 패치로 나누고, 각 패치의 Linear Embedding에 순서를 제공합니다.

▼ 3-2. Related Work

▼ CNN은 이미지 분류, 객체 검출, 세그멘테이션 등 다양한 비전 태스크에서 활용되고 있으나, 복잡한 아키텍처와 많은 연산이 필요합니다.

- CNN은 이미지 하나를 한 번에 인식하지만, ViT의 경우 이미지를 패치 단위로 쪼개기 때문에 Inductive Bias를 줄일 수 있습니다.

▼ the model of Cordonnier et al. (2020)

- 논문에서는 ViT와 가장 유사한 Paper라고 언급되어 있습니다.
- 입력 이미지로 부터 $2 * 2$ 패치 사이즈를 추출하고, full self-attention을 적용합니다.
- 그리고 ViT에 비해 패치 사이즈가 작아, small-resolution 이미지에만 적용 가능합니다. (단점)

▼ 기존 CNN과 self-attention을 결합한 모델의 경우 많은 연산이 요구되며, 이전에 이러한 연구들이 있었다고 ㅎ

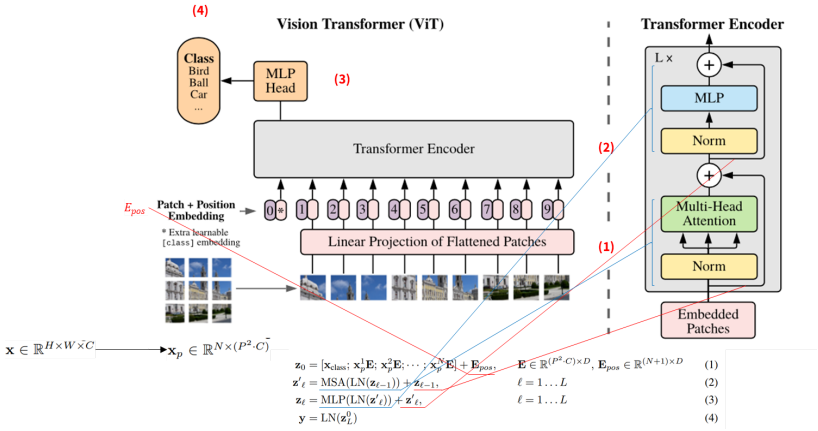
- by augmenting feature maps for image classification (Bello et al., 2019) or by further processing the output of a CNN using self-attention, e.g. for object detection (Hu et al., 2018;
  Carion et al., 2020)
- video processing (Wang et al., 2018; Sun et al., 2019), image classification (Wu et al., 2020)
- unsupervised object discovery (Locatello et al., 2020), or unified text-vision tasks (Chen
  et al., 2020c; Lu et al., 2019; Li et al., 2019)

▼ image GPT (iGPT) (Chen et al., 2020a)

- 이미지 해상도와 색 공간을 줄인 후 이미지 픽셀에 트랜스포머를 적용합니다.
- ImageNet에서 72%의 최대 정확도를 달성했습니다.

▼ 3-3. Method

▼ Vision Transformer(ViT)



- 직접 만든 이미지입니다.
- ViT(Vision Transformer)는 이미지를 여러 개의 작은 패치로 나눈 후, 이러한 패치를 Transformer 모델에 입력으로 제공하는 방식을 사용합니다. 각 패치는 일렬로 펼쳐진 벡터로 변환되며, 그러한 패치들은 시퀀스로 취급됩니다. 이 시퀀스는 Transformer의 인코더를 통과하고, 그 결과는 이미지의 글로벌한 특성을 포착하는 것이 가능합니다.
- ViT는 또한 위치 정보를 잃지 않기 위해 각 패치에 위치 임베딩(Positional Embedding)을 추가합니다. 이는 자연어 처리에서 문장 내 단어의 순서를 고려하는 것과 유사한 역할을 합니다.

▼ Fine-Tuning and Higher Resolution

- ViT는 대량의 데이터와 계산 능력을 필요로 하는 모델이기 때문에, 미리 훈련된 모델을 다른 작업이나 더 작은 데이터셋에 적용(fine-tuning)하기가 일반적입니다. Fine-tuning은 사전에 훈련된 모델의 가중치를 초기값으로 사용하고, 특정 작업에 대한 성능을 높이기 위해 추가로 훈련을 수행하는 과정입니다.
- 또한, 높은 해상도의 이미지를 처리하기 위해서는 더 큰 패치 크기나 더 많은 레이어를 추가할 수 있습니다. 이러한 변화는 모델의 파라미터 수를 늘리고, 그에 따라 계산 복잡성도 증가시키지만, 성능 향상을 가져올 수 있습니다.

▼ 3-4. Experiments

▼ SetUp

▼ Comparison to SOTA

▼ PreTraining Data Requirements

▼ Scaling Study

▼ Inspecting Vision Transformer

→ 시간이 없어서 정리를 못했습니다…

▼ 3-5. Conclusion

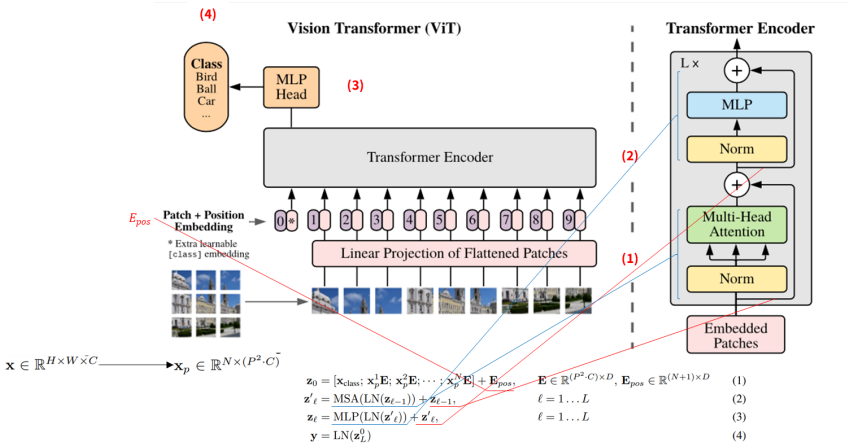
- 이미지 분류 작업에서도 Transformer는 뛰어난 성능을 보이며,
  이를 통해 다양한 비전 task에 Transformer를 활용할 수 있을 것이라는 가능성을 제시합니다.

---

4. Code

- 기존에 제공해주신 링크의 경우, jax로 작성되어 있어 익숙하지 않아 PyTorch 기반으로 코드를 탐색했습니다.
- 코드를 자세하게 분석하고 작성하기에 시간이 부족하여, 모델 아키텍처와 코드를 매칭 시켜 이해한대로 편집하였습니다.
- 코드 링크

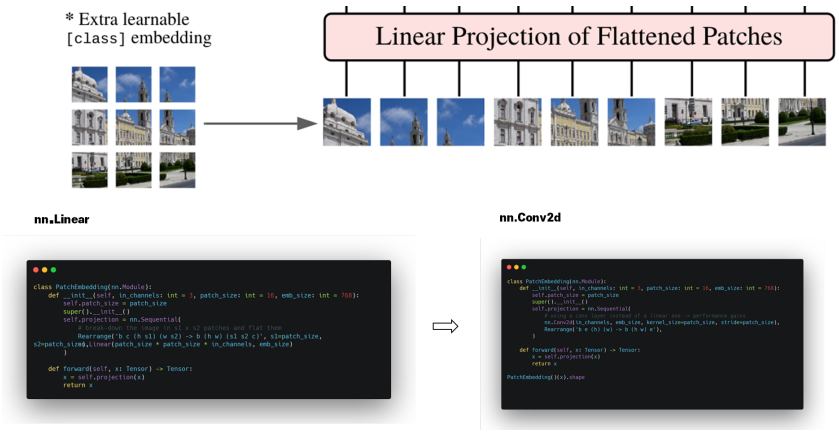
▼ Full Architecture



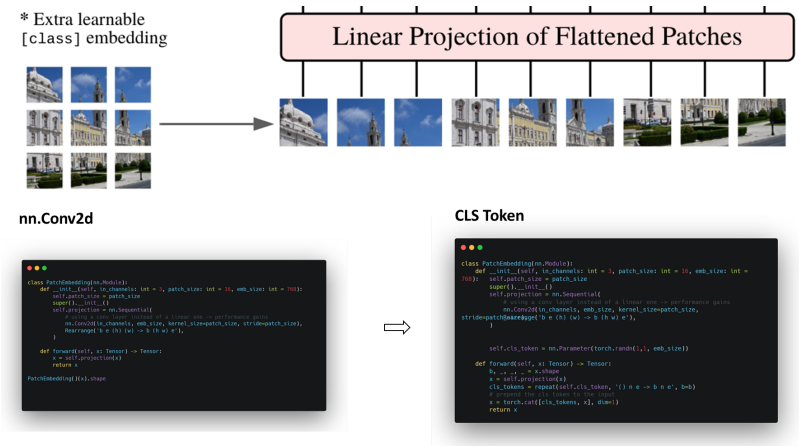
- Patches Embeddings
  - CLS Token
  - Position Embedding
- Transformer
  - Attention
  - Residuals
  - MLP
  - Transformer Encoder Block
- Head
- VIT

▼ Patches Embeddings

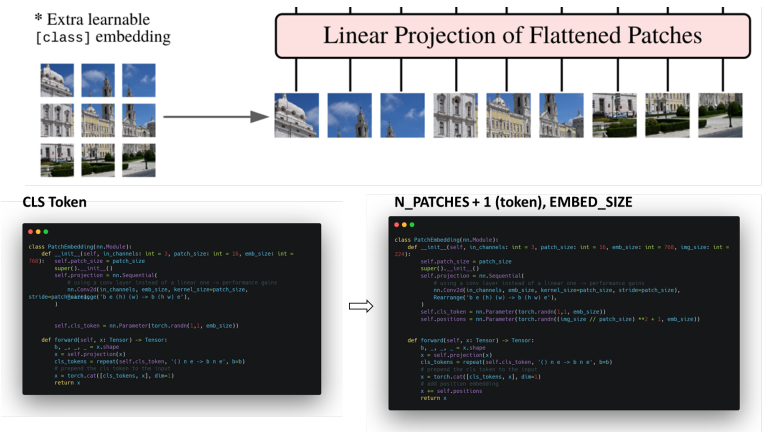
```
# resize to imagenet size
transform = Compose([Resize((224, 224)), ToTensor()])
x = transform(img)
x = x.unsqueeze(0) # add batch dim

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



▼ CLS Token

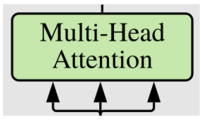


▼ Position Embedding



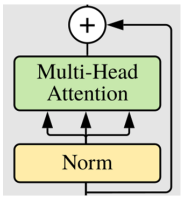
▼ Transformer Encoder

▼ Attention



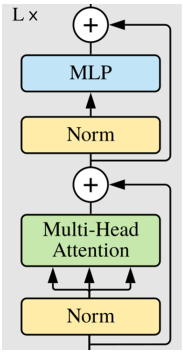

```python
class MultiHeadAttention(nn.Module):
    def __init__(self, emb_size: int = 768, num_heads: int = 8, dropout: float = 0):
        super().__init__()
        self.emb_size = emb_size
        self.num_heads = num_heads
        # fuse the queries, keys and values in one matrix
        self.qkv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

▼ Residuals



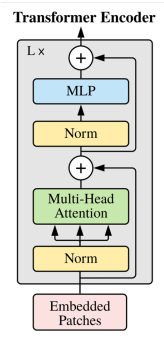

```python
class ResidualAdd(nn.Module):
    def __init__(self, fn):
        super().__init__()
        self.fn = fn

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

▼ Feed-forward




```python
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),
        )
```

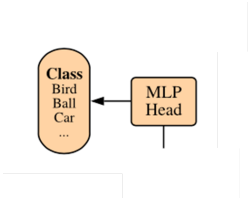
▼ Transformer Encoder Block




```python
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)
            )
            ))
```

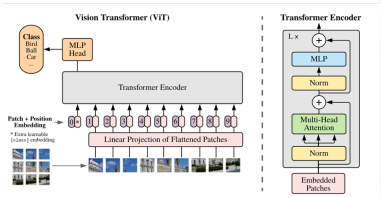
```python
class TransformerEncoder(nn.Sequential):
    def __init__(self, depth: int = 12, **kwargs):
        super().__init__(*[TransformerEncoderBlock(**kwargs) for _ in range(depth)])
```

▼ Head




```python
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))
```

▼ ViT = Patches Embeddings + Transformer Encoder + Head




```python
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)
        )
```

- FineTuning 예시 코드

---

5. 추후 방향성

- 최근 txt→img, txt+img → img 관련 생성 모델들을 이해하기 위해서 거쳐야하는 모델 중 하나인 CLIP이라는 모델이 있습니다.
- 텍스트와 이미지 데이터의 임베딩 간의 연결을 어떻게 할 것인가
- CLIP을 사용해 본 적은 있지만, 깊게 이해하기 위해 ViT이후 추가적으로 논문을 탐색하고자 합니다.

---

6. 참고 링크

- 원문

  https://arxiv.org/pdf/2010.11929.pdf

- 이외 참고한 링크는 북마크로 내용 사이에 첨부되어 있습니다.

---

7. ViT 이전에 참고할 만한 논문

- “Attention is all you need”논문만 읽어본 적이 있다면, ViT는 쉽게 이해할 수 있어서 추가적인 논문을 남기지 않았습니다.

  https://arxiv.org/pdf/1706.03762.pdf

---

8. ViT이후 관련 논문 for CLIP

1. **멀티모달 러닝**
   - 논문: "Multimodal Deep Learning" (링크)
   - 참고 자료: "A Survey on Multimodal Learning" (링크)
2. **Contrastive Learning**

- 논문: "SimCLR: A Simple Framework for Contrastive Learning of Visual Representations" (링크)
- 참고 자료: "Understanding Contrastive Representation Learning through Alignment and Uniformity on the Hypersphere" (링크)

3. **Zero-Shot Learning**
   - 논문: "Zero-Shot Learning - A Comprehensive Evaluation of the Good, the Bad and the Ugly" (링크)
   - 참고 자료: "Zero-Shot Learning in Modern NLP" (링크)
4. **Text Embedding**
   - 논문: "Efficient Estimation of Word Representations in Vector Space" (Word2Vec) (링크)
   - 참고 자료: "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding" (링크), T5
5. **Image Retrieval**
   - 논문: "Learning to Rank for Content-Based Image Retrieval" (링크)
   - 참고 자료: "Large-Scale Image Retrieval with Attentive Deep Local Features" (링크)

---

9. 리뷰 후기

NLP만 주로 다루다가 이미지 생성 모델로 넘어가면서, 순수 Vision 분야의 논문을 자세하게 읽어본 것은 이번이 처음이었습니다.
Transformer를 그대로 유지하려고 노력한 논문이기에, 기존에 Transformer를 이해하고 있어 읽기 편했고
실험 관련 내용을 자세하게 작성하지 못해서, 추후 마무리해서 완성하도록 하겠습니다.