Chapter 6. Transformers

Outline

A. From recurrence (RNN) to Attention-Based NLP Models

B. Self-Attention

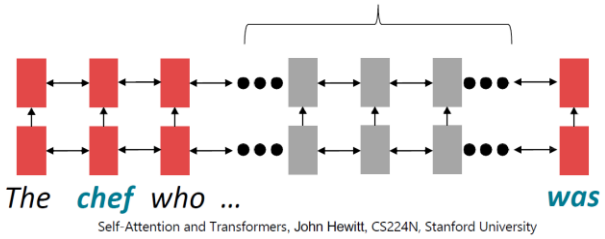
A. From recurrence (RNN) to Attention-Based NLP Models

1) Recurrent model의 Issue

Recurrent 모델 (RNN, Seq2Seq, etc..)은 모두 입력으로 Sequence를 받는다. 다만, 한 번에 모든 sequence를 처리하는 것이 아닌 순차적으로 입력 시퀀스를 처리한다. 이를 Linear Interaction Distance라고도 한다.

다시 말해, RNN은 'Left-to-Right' 형태로 입력을 처리하기 때문에 모델이 기본적으로 linear locality를 가진다. 이는 heuristic하다는 관점에서는 좋지만, Time complexity 입장에서는 안 좋다. Time complexity로 O(Sequence Length)를 요구하기 때문이다.

Problem: RNNs take **O(sequence length)** steps for distant word pairs to interact.



Self-Attention and Transformers, John Hewitt, CS224N, Stanford University

(1) Long term dependency

이렇게 Sequence 형태의 입력을 순차적으로 처리할 경우 뒤의 입력을 처리할 때 앞의 정보를 유실하는 Long term dependency가 발생한다.

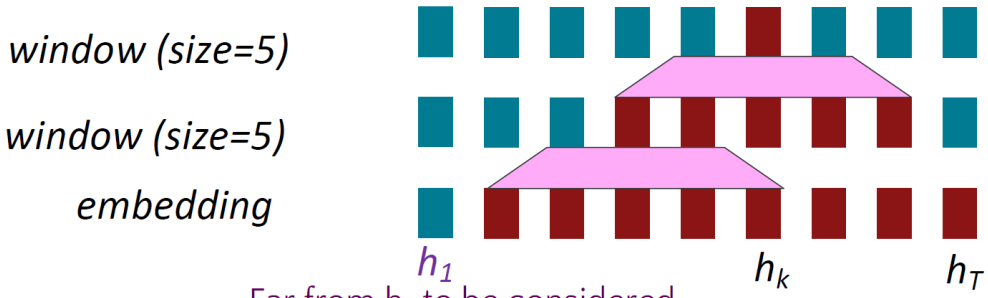
(2) Lack of Parallelizability

또한 순차적인 모델은 forward와 backward pass 모두에서 병렬 처리가 불가능하다. 따라서 Time Complexity가 줄일 수 없고, large dataset에 대해서는 다루기 힘들다.

2) Word Windows

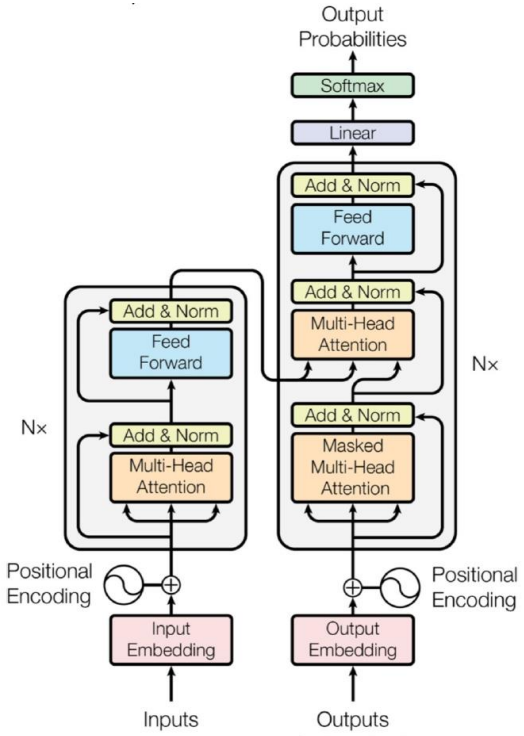
Word Window 모델은 앞서 말한 RNN 모델의 단점을 어느정도 해소할 수 있다. Word Window 모델은 local context를 aggregation한다. (1D Convolution) 병렬화 할 수 없는 operation이 sequence length를 증가시키지는 않는다.

또한 Long-term dependency또한 여전히 존재한다. Sequence 길이를 window size로 나눈만큼 time complexity를 가져가지만, sequence의 길이가 길어지면 무의미하다.



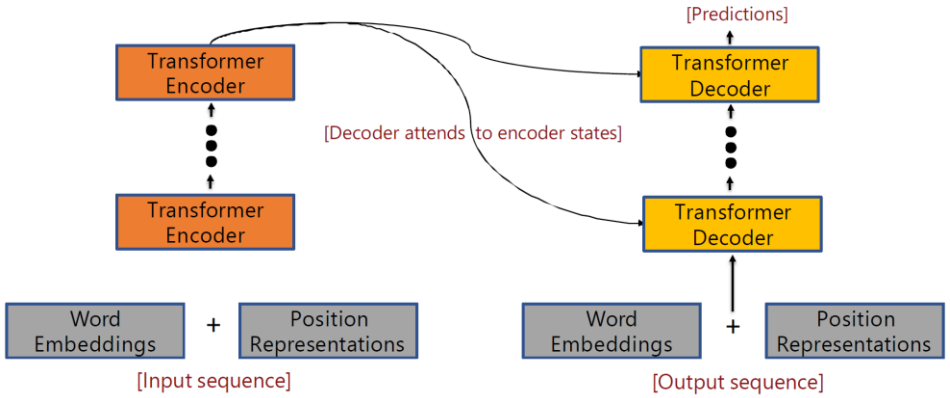
B. Transformer with Self-Attention

1) Transformer – Attention is all you need

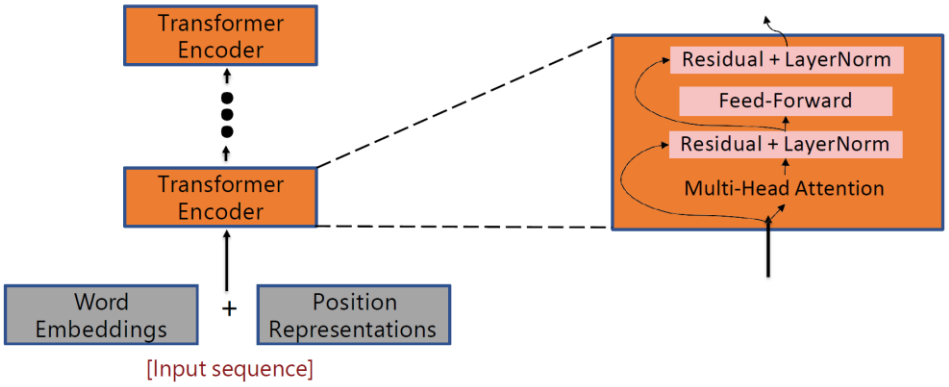


Transformer는 Attention 기반의 모델이다. 또한 Encoder-Decoder 모델이다.

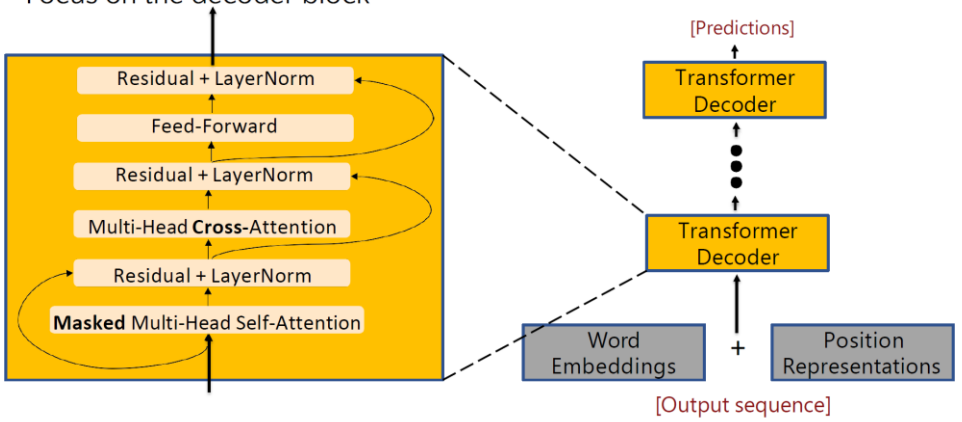
- Transformer Encoder and Decoder Blocks: no recurrence, no convolution



- Focus on the encoder block

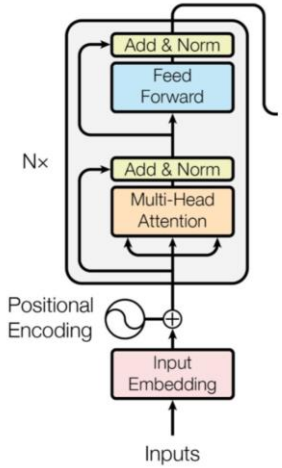


- Focus on the decoder block



2) Encoder block

Transformer input sequence는 input embedding으로 들어간다. Positional encoding을 더해서 최종적으로 인코더로 들어간다. 이 때, Mutli-head attention을 거쳐서 input sequence 내의 모든 token들의 attention을 구한 후 FFNN을 먹인다.



(1) Self-Attention (without learned weights)

Input: x_1, x_2, x_3 Output: y_1, y_2, y_3

(인코더의 입력을 만들어 내는 과정)

입력 시퀀스는 weighted sum된다.

$$y_i = \sum_j w_{ij} x_j$$

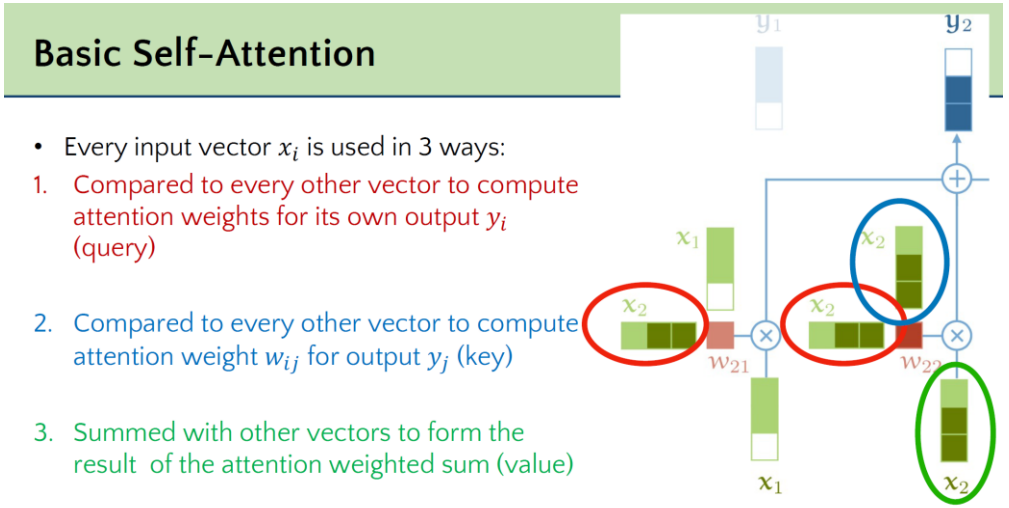
a. 이 때, weight는 학습되는 것이 아닌, x_i, x_j의 함수이다.

$$w'_{ij} = x_i^T x_j$$

b. 1부터 j까지 모두 summation한다.

$$w_{ij} = \frac{\exp w'_{ij}}{\sum_j \exp w'_{ij}}$$

이 부분에서는 weight를 학습하지 않는다. 즉 sequence의 순서가 결과 연산에 영향을 끼치지 않는다.(인코더 블록으로 들어가는 input embedding을 만들어 내는 것이 전부)



(2) Key-Query-Value Attention

$$k_i = W_k x_i \ , W_k \in R^{d \times d}$$

$$q_i = W_q x_i \ , W_q \in R^{d \times d}$$

$$v_i = W_v x_i \ , W_v \in R^{d \times d}$$

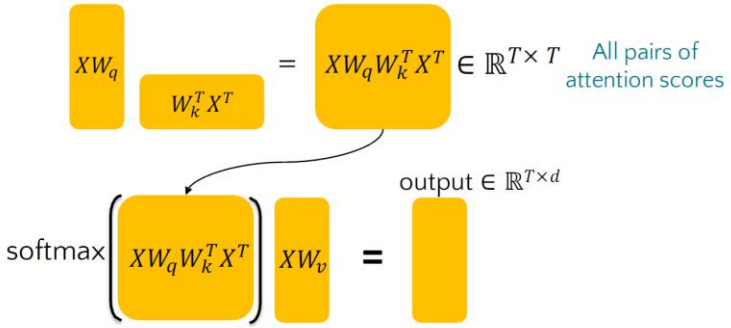
Query는 현재 시점의 Token으로 비교의 주체가 되는 subject이다.

Key는 비교하려는 대상으로 객체, object이다. 이 때 attention에서는 자신을 포함한 모든 토큰이 Key가 된다.

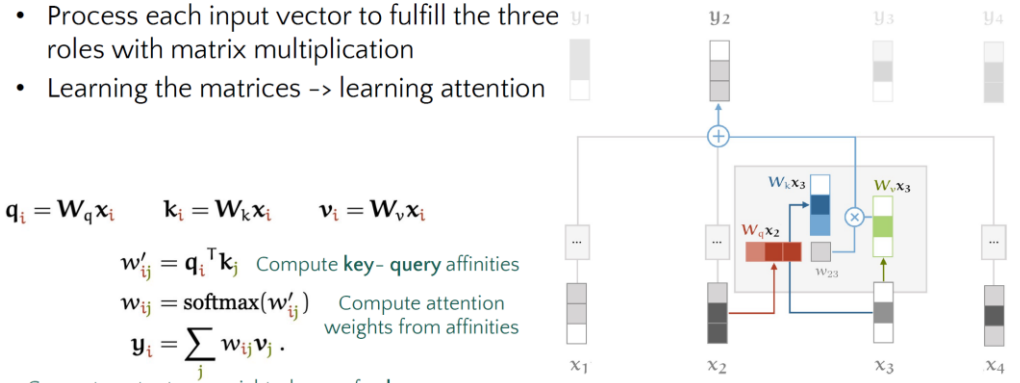
Value는 입력 시퀀스의 각 토큰과 관련된 실제 정보를 수치로 나타낸 실제 값으로 Key와 동일한 토큰을 지칭한다.

- if Query = 'I'
  - Key = 'I', 'am', 'a', 'teacher'
  - Query-key의 사이의 연관성을 구한다 = Attention

- Given the concatenation of input vectors $X = [x_1; \dots; x_T] \in \mathbb{R}^{T \times d}$.
- First, $XW_k \in \mathbb{R}^{T \times d}$, $XW_q \in \mathbb{R}^{T \times d}$, $XW_v \in \mathbb{R}^{T \times d}$.
- output = $\text{softmax}\left(XW_q(XW_k)^T\right) \cdot XW_v$



- Process each input vector to fulfill the three roles with matrix multiplication
- Learning the matrices -> learning attention

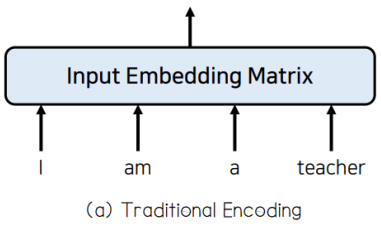


http://www.peterbloem.nl/blog/transformers

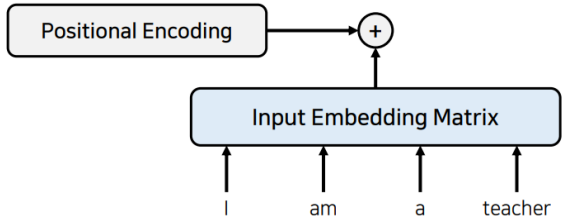
19

#1 하지만, 이런 Self-Attention에서도 문제점이 존재한다. 바로 Input sequence의 순서에 대한 정보가 결여된다는 것이다.

이를 위해 Positional Encoding을 같이 더해 인코더의 입력으로 넘겨준다.



(a) Traditional Encoding



(b) Transformer Encoding = Positional Encoding

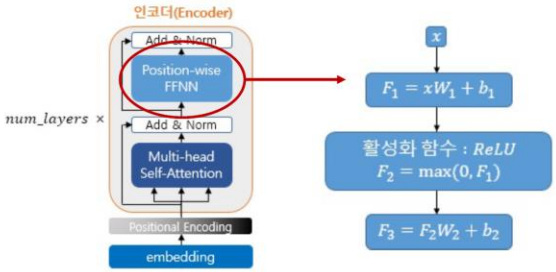
$$PE_{(pos,2i)} = sin(pos/10000^{2i/d_{model}})$$
$$PE_{(pos,2i+1)} = cos(pos/10000^{2i/d_{model}})$$

#2 또한 deep learning을 위한 nonlinearity가 없다는 점이다. Self-attention을 여전히 weighted average에 불과하다.

이를 위해 Feedforward network를 각 self-attention output에 추가해준다.

No elementwise nonlinearities in self-attention
Stacking more self-attention layers just re-averages **value** vectors
Fix: Add a position-wise **feed-forward network (FFNN)** to post-process each output vector.
FFNN network processes the result of attention

$$\mathbf{m}_i = MLP(\mathbf{x}_i) = ReLU(\mathbf{x}_i\mathbf{W}_1 + \mathbf{b}_1)\,\mathbf{W}_2 + \mathbf{b}_2$$

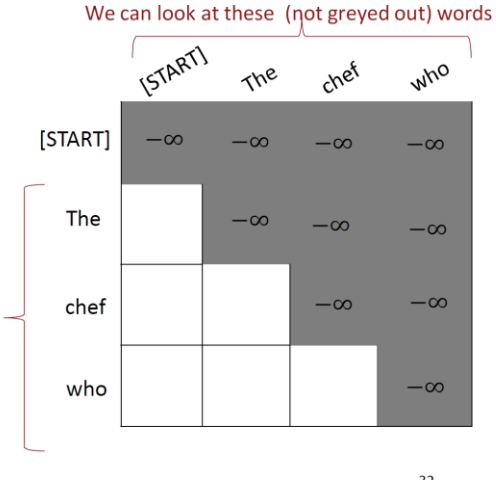


#3 마지막으로 시퀀스를 처리할 때 미래의 값을 미리 보지 않는다는 보장을 해줘야 한다.
(Need to ensure that we don't "look at the future" when predicting a sequence)

이를 위해 Masking을 통해 Future 값에 대한 정보를 지워버린다.
(Mask out the future by artificially setting attention weights to 0.)

- To use self-attention in **decoders**, we must not peek at the future.
- At every timestep, we could change the set of **keys and queries** to include only past words. (Inefficient!)
- To enable parallelization, we **mask out attention** to future words by setting attention scores to $-\infty$.

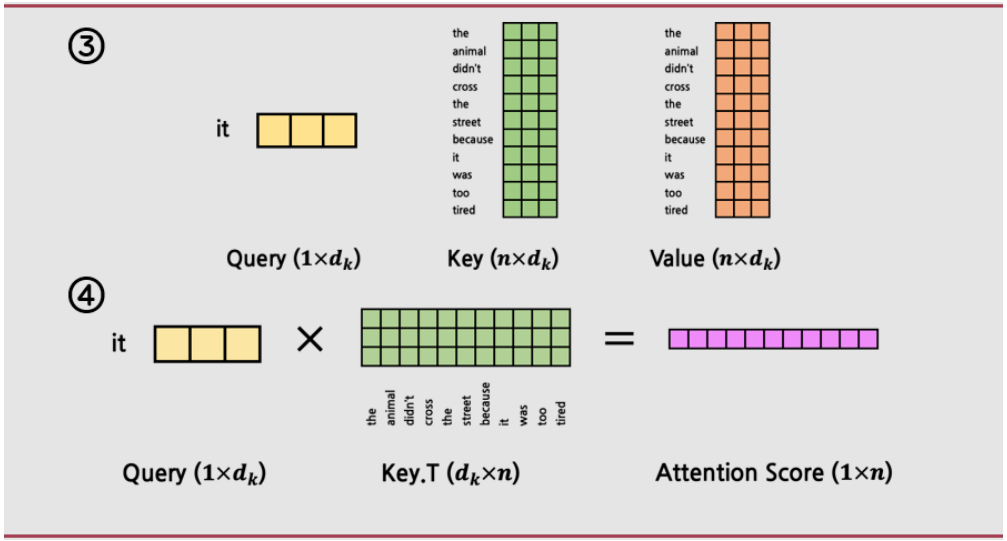
$$e_{ij} = \begin{cases} q_i^T k_j, & if\ j < i \\ -\infty, & otherwise \end{cases}$$

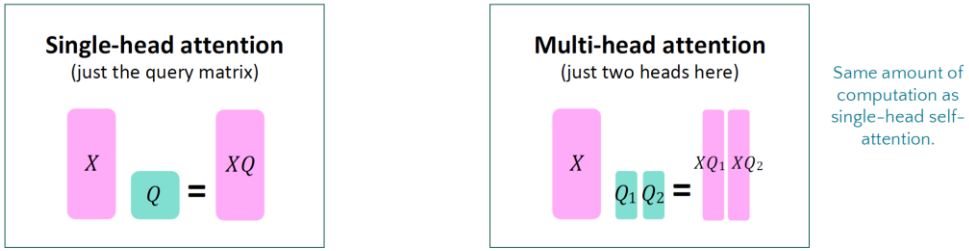


(3) Multi-Head Self-Attention

트랜스포머의 경우 Attention 중 Scaled dot product attention을 수행한다. Attention의 목적은 토큰들이 서로 얼마나 더 큰 영향력을 가지고 있는지 보기 위함이다. 이 때, Self-Attention이라는 것은 한 문장 내에서 토큰들의 attention을 구하는 것이고, Cross-Attention이라는 것은 서로 다른 문장에서 토큰들의 attention을 구하는 것이다.

먼저 Query, Key, Value값들은 입력으로 들어오는 Token embedding을 Fully connected layer에 넣어 생성된다. 이 값들은 모두 서로 다르고, 이 세 개의 임베딩 벡터(matrix)를 만들기 위해 3개의 서로 다른 FC layer가 존재한다.



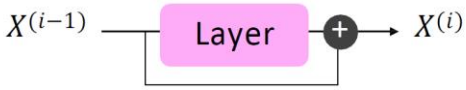


38

입력 시퀀스를 한 번에 하나의 시퀀스로 정재해서 처리하는 것이 아닌, Multi-head attention을 통해 여러 개의 head를 가지고 연산을 하며 이렇게 연산된 결과는 각각 다른 weight을 가지게 된다.

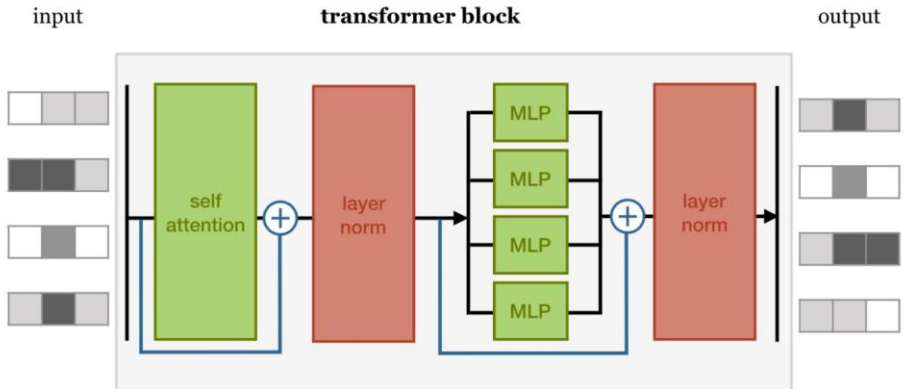
또한 Residual connection을 통해 모델의 표현력을 좀 더 높인다.

- We let $X^{(i)} = X^{(i-1)} + \text{Layer}(X^{i-1})$ (so we learn only "the residual" from the previous layer)



그 후 normalization을 통해 값을 안정화시킨다.

Self-attention layer -> **Layer normalization** -> Dense layer



Layer Normalization은 모델이 학습을 빠르게 할 수 있도록 도와주는 trick이다. 각 layer에 평균과 분산값을 이용해 처리한다.

LayerNorm's success may be due to its normalizing gradients [Xu et al., 2019]

Let $x \in \mathbb{R}^d$ be an individual (word) vector in the model.

Let $\mu = \Sigma_{j=1}^{d} x_j$; which is mean; $\mu \in \mathbb{R}$.

Let $\sigma = \sqrt{\frac{1}{d}\Sigma_{j=1}^{d}(x_j - \mu)^2}$; this is the standard deviation; $\sigma \in \mathbb{R}$.

Let $\gamma \in \mathbb{R}^d$ and $\beta \in \mathbb{R}^d$ be learned "gain" and "bias" parameters.

Then layer normalization computes:

$$\text{output} = \frac{x - \mu}{\sqrt{\sigma + \epsilon}}$$

Normalize by scalar mean and variance

Modulate by learned elementwise gain and bias

Scaled Dot product attention

- Instead of the self-attention function we've seen:

$$\text{output}_\ell = \text{softmax}(XQ_\ell K_l^T X^\top) * XV_\ell$$

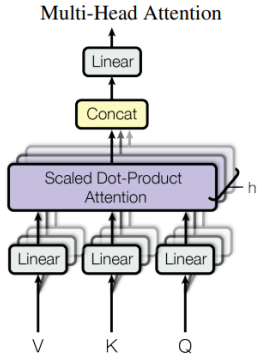
- Divide the attention scores by d/h, to stop the scores from becoming large just as a function of d/h (The dimensionality divided by the number of heads.)

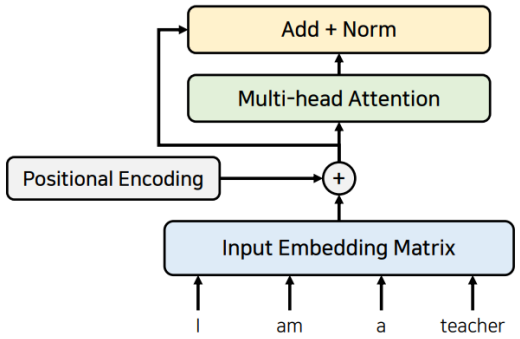
$$\text{output}_\ell = \text{softmax}\left(\frac{XQ_\ell K_\ell^\top X^\top}{\sqrt{d/h}}\right) * XV_\ell$$

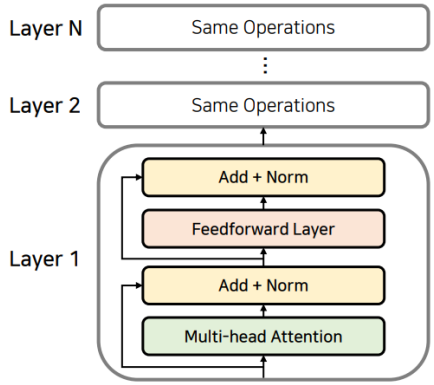
43

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

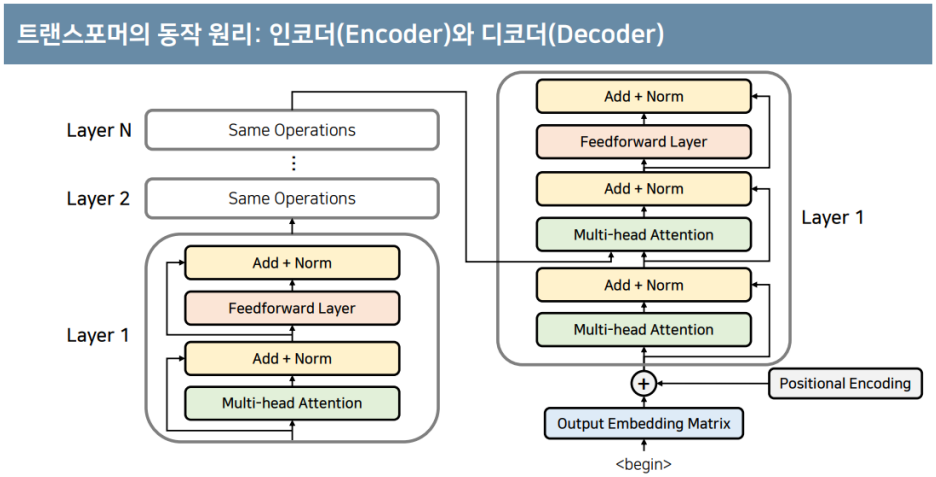
$$W_i^Q \in \mathbb{R}^{d_{\text{model}} \times d_k}, W_i^K \in \mathbb{R}^{d_{\text{model}} \times d_k}, W_i^V \in \mathbb{R}^{d_{\text{model}} \times d_v}, W^O \in \mathbb{R}^{hd_v \times d_{\text{model}}}$$





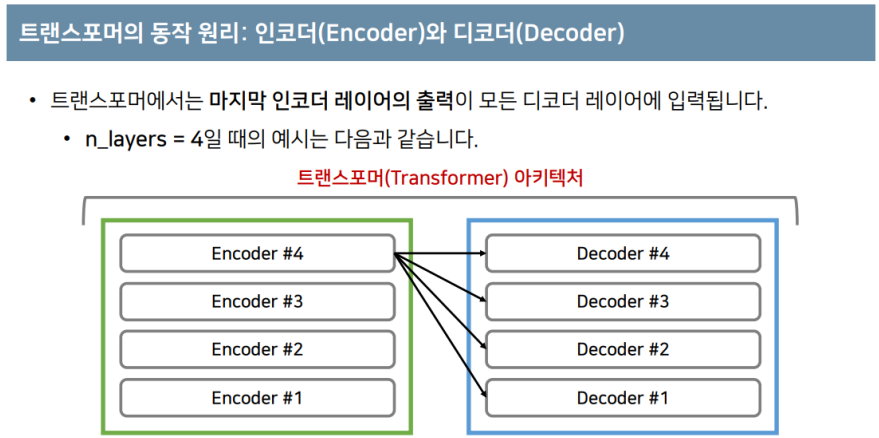


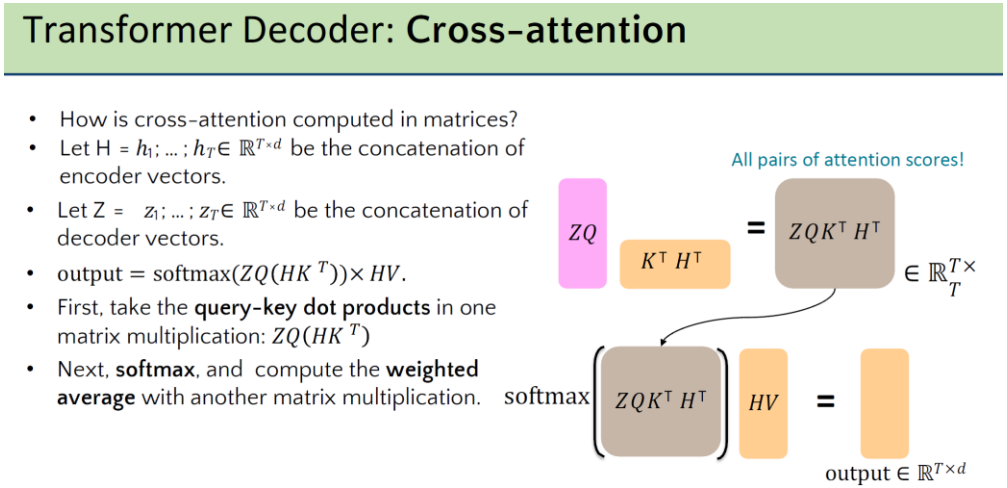
3) Decoder Block



Transformer의 decoder에서는 이전 step의 출력 sequence를 self-attention을 통해 입력을 받고, positional encoding과 더해져 입력으로 입력으로 넘어간다.

Decoder에서 가장 큰 특징은 바로 Cross-Attention을 한다는 것이다. Cross Attention은 앞서 말했듯, 서로 다른 문장의 토큰들에 대한 attention을 구하는 것이다. 그러면 이렇게 이전 출력 시퀀스의 임베딩 말고 다른 문장이 될 수 있는 것은, 바로 인코더의 최종 출력값이다. 즉, 트랜스포머에서는 마지막 인코더 레이어의 출력이 모든 디코더 레이어에 입력되어 Cross Attention을 진행하게 된다.





4) Transformer의 단점

a. Quadratic Compute in self-attention

b. Position representation

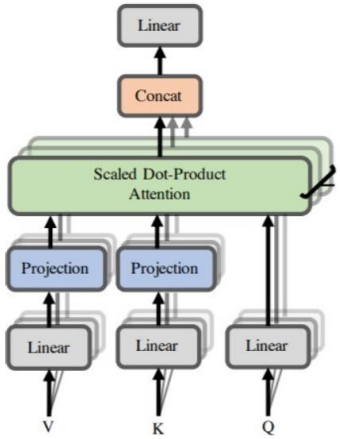
일단, attention의 경우 나와 모든 대상에 대해서 attention score를 구하므로, 연산량이 매우 많다. 또한, 이렇게 모든 토큰들에 대해서 attention을 진행하다 보니, 위치 관계가 제대로 반영되지 않는다는 문제점이 있다.

C. Efficient Transformers

1) Linformer: Self-attention with Linear Complexity

기존 Transformer의 경우 Self-attention 연산을 하는데 매우 큰 계산 복잡도를 요한다. (O(|n^2|) 이러한 문제점을 극복하기위해 나온 모델이 바로 Linformer이다.

Key idea: map the sequence length dimension to a lower-dimensional space for values, keys



기존의 Transformer에서 Self-Attention은 아래와 같다.

- $\text{MultiHead}(Q,K,V) = Concat(head_1, head_2, \dots, head_h)W^O$ where
  $Q,K,V \in R^{n\times d_m}$: input embedding matrices,
  n: sequence length,
  d_m: the embedding dimension,
  h: the number of heads

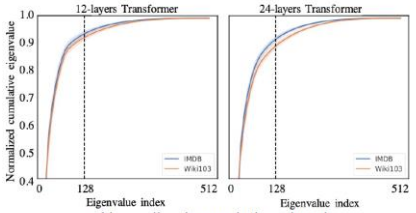
- Each head

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

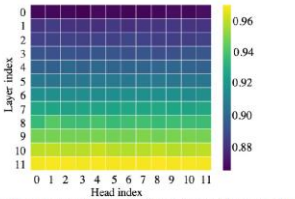
where
$W_i^Q, W_i^K \in \mathbb{R}^{d_m\times d_k}, W_i^V \in \mathbb{R}^{d_m\times d_v}, W^O \in \mathbb{R}^{hd_v\times d_m}$: learned matrices,
d_k, d_v: the hidden dimensions of the projection subspaces.

Linformer에서 지적하는 것은 Self-Attention은 Low rank라는 것이다.

Singular value decomposition(SVD)을 attention hit-map matrix에 적용해 eigen value값을 구하고, 그 값들을 큰값부터 순서대로 누적하며 나타냈을 때 모델 capacity가 어떤지를 볼 수 있는 것이다.



Normalized cumulative singular value averaged over 10k sentences.



Normalized cumulative singular value at the 128-th largest singular value (out of 512)

Theorem 1. (self-attention is low rank) *For any* $Q,K,V \in \mathbb{R}^{n\times d}$ *and* $W_i^Q, W_i^K, W_i^V \in \mathbb{R}^{d\times d}$, *for any column vector* $w \in \mathbb{R}^n$ *of matrix* VW_i^V, *there exists a low-rank matrix* $\tilde{P} \in \mathbb{R}^{n\times n}$ *such that*

$$\Pr(\|\tilde{P}w^T - Pw^T\| < \epsilon\|Pw^T\|) > 1 - o(1) \text{ and } rank(\tilde{P}) = \Theta(\log(n)), \quad (3)$$

where the context mapping matrix P *is defined in (2).*

- $P = \text{softmax}\left[\frac{QW_i^Q(KW_i^K)^T}{\sqrt{d_k}}\right] = \exp(A)\cdot D_A^{-1}$ where D_A is $n\times n$ diagonal matrix.
- $\tilde{P} = \exp(A)\cdot D_A^{-1}R^TR$ where $R^{k\times n}$ with i.i.d. from $N(0,1/k)$ for $k = 5logn/(\epsilon^2-\epsilon^3)$
- **Approximate P with a low-rank matrix P_{low}**

$$P \approx P_{low} = \Sigma_{i=1}^k \sigma_i u_i v_i^T = [u_1,\dots,u_k]\, diag\{\sigma_1,\dots,\sigma_k\}\begin{bmatrix} v_1 \\ \vdots \\ v_k \end{bmatrix}$$

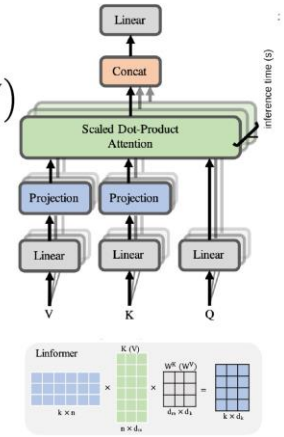
따라서 Linformer에서는 linear Self-Attention을 제시한다.

- Add two **linear projection matrices** $E_i, F_i \in \mathbb{R}^{n \times k}$.

$$\overline{head_i} = Attention\left(QW_i^Q, E_i K W_i^K, F_i V W_i^V\right)$$

$$= \text{softmax}\underbrace{\left[\frac{QW_i^Q\left(E_i K W_i^K\right)^T}{\sqrt{d_k}}\right]}_{\bar{P}: n \times k} \underbrace{F_i V W_i^V}_{k \times d}$$

- **Time and space**: $O(nk)$ where $k \ll n$



61

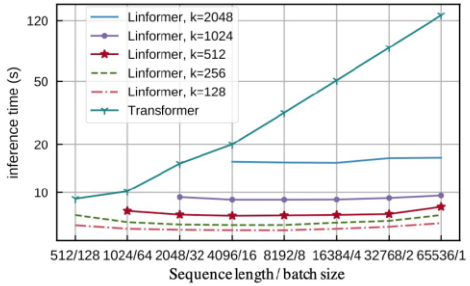
Theorem 2. *(Linear self-attention) For any $Q_i, K_i, V_i \in \mathbb{R}^{n \times d}$ and $W_i^Q, W_i^K, W_i^V \in \mathbb{R}^{d \times d}$, if $k = \min\{\Theta(9d\log(d)/\epsilon^2), 5\Theta(\log(n)/\epsilon^2)\}$, then there exists matrices $E_i, F_i \in \mathbb{R}^{n \times k}$ such that, for any row vector w of matrix $QW_i^Q(KW_i^K)^T/\sqrt{d}$, we have*

$$\Pr\left(\|\text{softmax}(wE_i^T)F_i V W_i^V - \text{softmax}(w)VW_i^V\| \le \epsilon\|\text{softmax}(w)\|\|VW_i^V\|\right) > 1 - o(1) \quad (8)$$

When $k = O(d/\epsilon^2)$ (independent of n), we can approximate $P \cdot VW_i^V$ using **linear self-attention** with ϵ error.

Experiment: Inference Time vs Sequence Length

Standard Transformer becomes slower at longer sequence lengths. Linformer speed remains relatively flat and is significantly faster at long sequences.



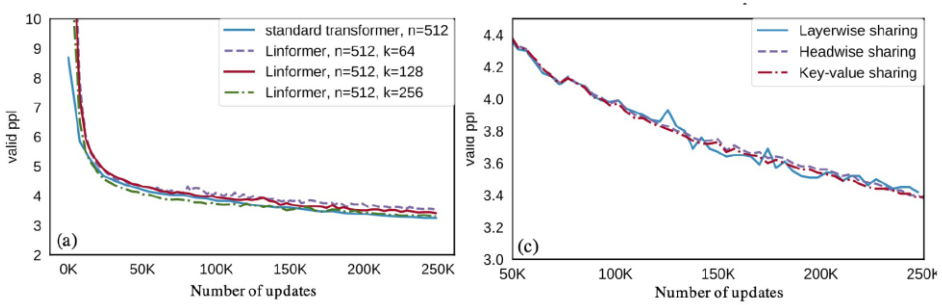
Parameter Sharing between projection

- Share parameters for the linear projection matrices E_i, F_i **across layers and heads**.

1. **Headwise sharing**: for each layer, share two projection matrices E and F s.t. $E_i = E$ and $F_i = F$ across all heads i.
2. **Key-value sharing**: do headwise sharing, and for each layer, create a single projection matrix E s.t. $E_i = F_i = E$ for each key-value projection matrix across all head i.
3. **Layerwise sharing**: use single projection matrix E across all layers, for all heads, and for both key and value.

Experiment: Pretraining Perplexities

Effect of projected dimension: Linformer performs better as projected dimension k increases.
Effect of sharing projections: the validation perplexity of layerwise sharing almost matche that of the non-shared model.



Experiment: Inference Time Efficiency

Inference time/memory efficiency of Linformer against standard Transformer

- For n = 512 and k = 128, Linformer has 1.5× faster inference time, and allows for a 1.7× larger maximum batch size than the Transformer.

length n	projected dimensions k				
	128	256	512	1024	2048
512	1.5x	1.3x	-	-	-
1024	1.7x	1.6x	1.3x	-	-
2048	2.6x	2.4x	2.1x	1.3x	-
4096	3.4x	3.2x	2.8x	2.2x	1.3x
8192	5.5x	5.0x	4.4x	3.5x	2.1x
16384	8.6x	7.8x	7.0x	5.6x	3.3x
32768	13x	12x	11x	8.8x	5.0x
65536	20x	18x	16x	14x	7.9x

length n	projected dimensions k				
	128	256	512	1024	2048
512	1.7x	1.5x	-	-	-
1024	3.0x	2.9x	1.8x	-	-
2048	6.1x	5.6x	3.6x	2.0x	-
4096	14x	13x	8.3x	4.3x	2.3x
8192	28x	26x	17x	8.5x	4.5x
16384	56x	48x	32x	16x	8x
32768	56x	48x	36x	18x	16x
65536	60x	52x	40x	20x	18x

Table 3: Inference-time efficiency improvements of the Linformer over the Transformer, across various projected dimensions k and sequence lengths n. Left table shows time saved. Right table shows memory saved.

Experiment: Downstream Results

Linformer has comparable downstream performance to RoBERTa.

- Linformer slightly outperforms RoBERTa at k=256.

Linformer's layerwise sharing strategy exhibits the best accuracy among three parameter sharing strategies.
Linformer pretrained with longer sequence length has similar results to the one pretrained with shorter length.

- Performance of Linformer is determined by the projected dimension k instead of the ratio n/k.

n	Model	SST-2	IMDB	QNLI	QQP	Average
512	Liu et al. (2019), RoBERTa-base	93.1	94.1	90.9	**90.9**	92.25
	Linformer, 128	92.4	94.0	90.4	90.2	91.75
	Linformer, 128, shared kv	**93.4**	93.4	90.3	90.3	91.85
	Linformer, 128, shared kv, layer	93.2	93.8	90.1	90.2	91.83
	Linformer, 256	93.2	94.0	90.6	90.5	92.08
	Linformer, 256, shared kv	93.3	93.6	90.6	90.6	92.03
	Linformer, 256, shared kv, layer	93.1	94.1	**91.2**	90.8	**92.30**
512	Devlin et al. (2019), BERT-base	92.7	93.5	91.8	89.6	91.90
	Sanh et al. (2019), Distilled BERT	91.3	92.8	89.2	88.5	90.45
1024	Linformer, 256	93.0	93.8	90.4	90.4	91.90
	Linformer, 256, shared kv	93.0	93.6	90.3	90.4	91.83
	Linformer, 256, shared kv, layer	93.2	**94.2**	90.8	90.5	92.18

Contribution & Limitation

Contribution

a. First theoretically proven linear-time transformer architecture

(Reduce the quadratic complexity of self-attention to the linear complexity)

b. Simple

(Two linear projections for query and key)

Limitations

a. Applicable only to encoders, not decoders

2) Performer: Rethinking Attention with Performers

(1) Problem

= Bottleneck of Transformer is self-attention (O(|n^2|)

따라서 Linformer와 같이 O(|n|)의 time complexity를 가져가기 위한 방법을 제안한 모델이다.

(2) Regular Attention Mechanism

For matrices $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{L\times d}$ where L is the size of an input sequence of tokens and d is the hidden dimension:

Bidirectional dot-product attention (in encoder self-attention and encoder-decoder attention)

$$Att_{\leftrightarrow}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \mathbf{D}^{-1}\mathbf{A}\mathbf{V}$$

where $\mathbf{A} = \exp(\mathbf{Q}\mathbf{K}^{\mathrm{T}}/\sqrt{d})$ and $\mathbf{D} = \mathrm{diag}(\mathbf{A}\mathbf{1}_L)$

($\mathbf{1}_L$ is all-one vector of length L, and

$\mathrm{diag}(\cdot)$ is a diagonal matrix with the input vector as the diagonal)

Unidirectional dot-product attention (in decoder)

$$Att_{\rightarrow}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \widetilde{\mathbf{D}}^{-1}\widetilde{\mathbf{A}}\mathbf{V}$$

where $\widetilde{\mathbf{A}} = tril(\mathbf{A})$ and $\widetilde{\mathbf{D}} = \mathrm{diag}(\widetilde{\mathbf{A}}\mathbf{1}_L)$

($tril(\cdot)$ returns the lower-triangular part of the argument matrix including diagonal)

(3) Generalized Kernelizable Attention

Consider attention $\mathbf{A} \in \mathbb{R}^{L\times L}$ as **kernel** $\mathrm{K}: \mathbb{R}^d \times \mathbb{R}^d \rightarrow \mathbb{R}_+$: $\mathbf{A}(i,j) = \mathrm{K}(\mathbf{q}_i^{\mathrm{T}}, \mathbf{k}_j^{\mathrm{T}})$

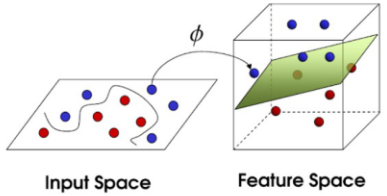
- $\mathbf{q}_i$ and $\mathbf{k}_j$ stands for the ith query row vector and jth key row vector, respectively.

Define **kernel** $\mathrm{K}(\mathbf{x}, \mathbf{y}) = \mathbb{E}[\phi(\mathbf{x})^{\mathrm{T}}\phi(\mathbf{y})]$ for mapping $\phi: \mathbb{R}^d \rightarrow \mathbb{R}_+^r$ $(r > 0)$

- $\phi(\mathbf{u})$: random feature map for $\mathbf{u} \in \mathbb{R}^d$

For matrices $\mathbf{Q}', \mathbf{K}' \in \mathbb{R}^{L\times r}$ with rows given as $\phi(\mathbf{q}_i^{\mathrm{T}})^{\mathrm{T}}$ and $\phi(\mathbf{k}_i^{\mathrm{T}})^{\mathrm{T}}$ respectively:
Approximate attention

$$\widehat{Att_{\leftrightarrow}}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \widehat{\mathbf{D}}^{-1}\left(\mathbf{Q}'\left((\mathbf{K}')^{\mathrm{T}}\mathbf{V}\right)\right) \text{ where } \widehat{\mathbf{D}} = \mathrm{diag}(\mathbf{Q}'((\mathbf{K}')^{\mathrm{T}}\mathbf{1}_L))$$



(4) Approximation of Regular Attention Mechanism

Approximation of **AV** (before $\mathbf{D}^{-1}$ renormalization) via feature maps.
Time: $O(Lrd)$
Space: $O(Lr + Ld + rd)$

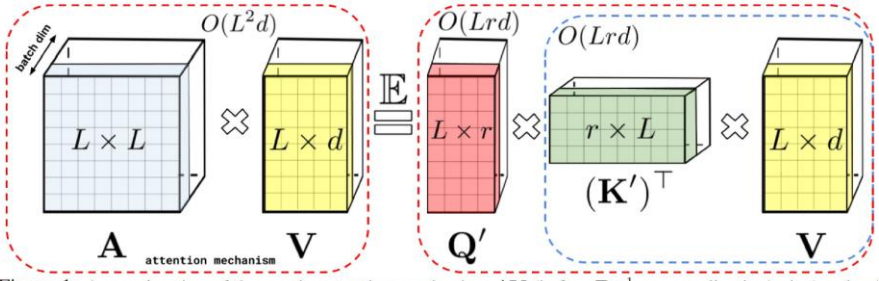


Figure 1: Approximation of the regular attention mechanism **AV** (before $\mathbf{D}^{-1}$-renormalization) via (random) feature maps. Dashed-blocks indicate order of computation with corresponding time complexities attached.

(5) How to Approximate Softmax-Kernels

Define **kernel** $\mathrm{K}(\mathbf{x},\mathbf{y})$ to be a softmax-kernel, e.g., $\mathrm{SM}(\mathbf{x},\mathbf{y}) \equiv \exp(\mathbf{x}^{\mathrm{T}}\mathbf{y})$

Fast Attention Via positive Orthogonal Random features (FAVOR+)

$$\widehat{\mathrm{SMREG}}^{+}_{\mathrm{m}}(\mathbf{x},\mathbf{y}) \equiv \mathbb{E}_{\omega\sim\mathrm{Unif}(\sqrt{d}\mathrm{S}^{d-1})}\left[\exp(\sqrt{d}\frac{\omega^{\mathrm{T}}}{\|\omega\|}\mathbf{x}-\frac{\|\mathbf{x}\|^2}{2})\exp(\sqrt{d}\frac{\omega^{\mathrm{T}}}{\|\omega\|}\mathbf{y}-\frac{\|\mathbf{y}\|^2}{2})\right]$$

$$\mathrm{K}(\mathbf{x},\mathbf{y}) = \phi(\mathbf{x})\cdot\phi(\mathbf{y})$$

Experiment: Computational Cost

Performer reaches nearly linear time and sub-quadratic memory consumption.
Performer achieves nearly optimal speedup and memory efficiency possible.

- "X"-line is the identity function simply returning the V-matrix.

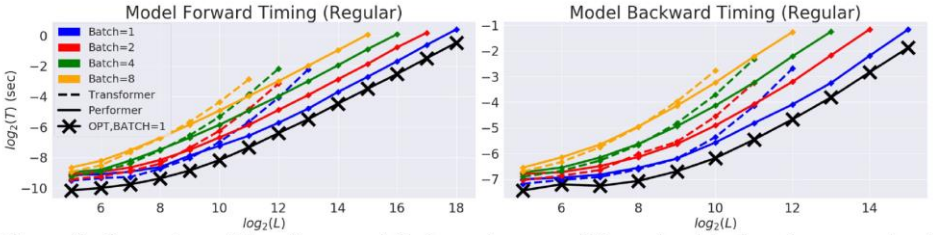


Figure 3: Comparison of Transformer and Performer in terms of forward and backward pass speed and maximum L allowed. "X" (OPT) denotes the maximum possible speedup achievable, when attention simply returns the $\mathbf{V}$-matrix. Plots shown up to when a model produces an out of memory error on a V100 GPU with 16GB. Vocabulary size used was 256. Best in color.

Experiment: Large Length Training

- On ImageNet64 with L=12288 (64×64×3) that is unfeasible for regular Transformer (left):
- Performer/6-layers matches Reformer/12-layers, and Performer/12-layers matches Reformer/24-layers.
- On TrEMBL with L=8192 (right):
- Performer train continuously to 24% while smaller Transformer is quickly bounded at 19%.

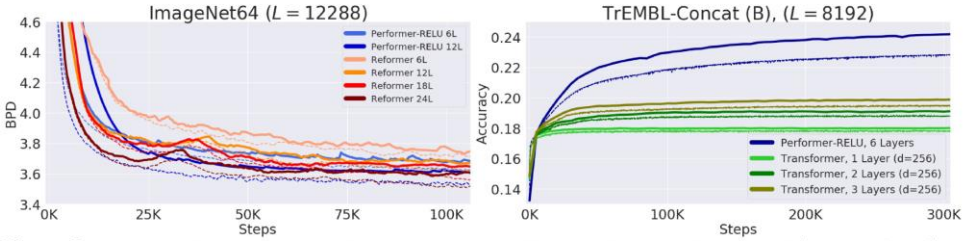


Figure 7: Train = Dashed, Validation = Solid. For ImageNet64, all models used the standard $(n_{heads}, d_{ff}, d) = (8, 2048, 512)$. We further show that our positive softmax approximation achieves the same performance as ReLU in Appendix D.2. For concatenated TrEMBL, we varied $n_{layers} \in \{1, 2, 3\}$ for the smaller Transformer. Hyperparameters can be found in Appendix A.

Contribution & Limitations

Contribution

a. Provably accurate and practical estimation of regular (softmax) full-rank attention

i) Linear space time complexity

ii) Without relying on any priors such as sparsity or low-rankness

b. Fully compatible with regular Transformers

i) Applicable to both encoders and decoders

Limitations

a. Unclear if this kernel paradigm is better than the learnable pattern or memory base models

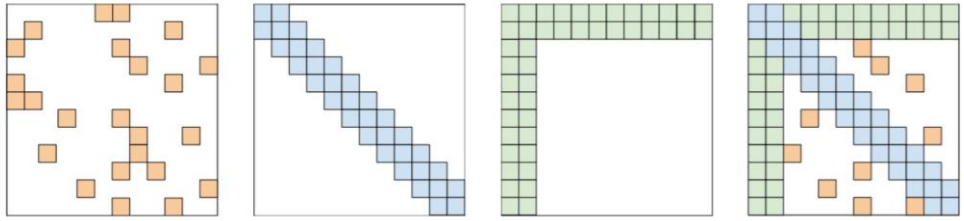
3) Big Bird: Transformers for Longer Sequence

(1) Recent work on improving on quadratic self-attention cost

Considerable recent work has gone into the question, *Can we build models like Transformers without paying the $O(T^2)$ all-pairs self-attention cost?*

- For example, **BigBird** [Zaheer et al., 2021]

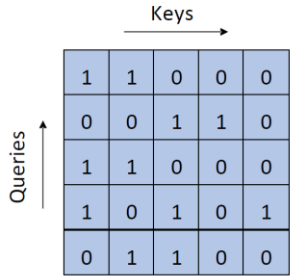
Key idea: replace all-pairs interactions with a family of other interactions, **like local windows**, **looking at everything**, and **random interactions**.



(a) Random attention (b) Window attention (c) Global Attention (d) BIGBIRD

Adjacency Matrix A(i, j) = 1

Query-i and Key-j



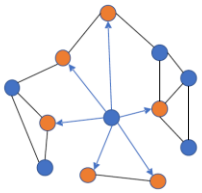
(2) Generalized Attention Mechanism

Self-attention can be generalized to a graph.

$\boldsymbol{X} = (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n) \in \mathbb{R}^{n \times d}$

D: Directed graph with vertex set is [n] = {1, ..., n}

$N(i)$: Out-neighbors set of node i



$$\text{ATTN}_D(\boldsymbol{X})_i = \boldsymbol{x}_i + \sum_{h=1}^{H} \sigma\left(Q_h(\boldsymbol{x}_i) K_h(\boldsymbol{X}_{N(i)})^T\right) \cdot V_h(\boldsymbol{X}_{N(i)})$$

$Q_h, K_h : \mathbb{R}^d \to \mathbb{R}^m$ $\quad V_h : \mathbb{R}^d \to \mathbb{R}^d$ $\quad X_{N(i)}$: Stacking $\{\boldsymbol{x}_j : j \in N(i)\}$

(3) Sparse Attention

Graph sparsification problem: Random graphs can approximate complete graphs (expanders)

Intuition

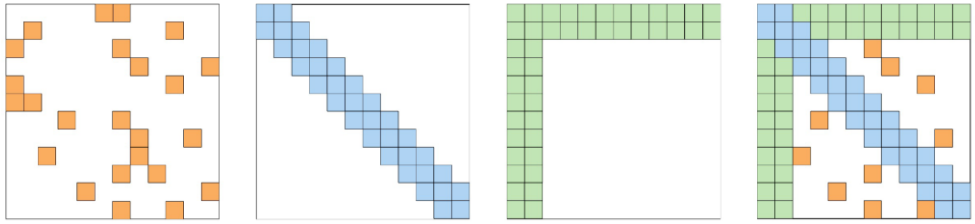
Erdős-Rényi model: Choose edges independently with some prob.

Small word property: with Θ(n) edges, the avg. length of the shortest path is logarithmic Θ(logn)

Approximates complete graph.

(4) Big Bird

- **Random Attention + Local Attention + Global Attention**

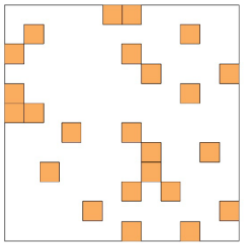


(a) Random attention (b) Window attention (c) Global Attention (d) BIGBIRD

Big Bird: Transformers for Longer Sequences [Zaheer *et al.*, 2021]

a. Random Attention

- **Random Attention**
  - Each query attends over **r** randomly chosen keys: A(i, ·) = 1.
  - Information can flow fast between any pair of nodes (rapid mixing time for random walks).
  - A(i,.) = 1 for r keys.



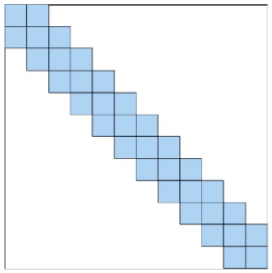
(a) Random attention

b. Window local attention

Window local attention

- Each token attends to a set of local neighboring tokens (w).
- Each query block with index j attends to the key block with index j – (w – 1)/2 to j + (w – 1)/2, including key block j.

$$A(i, i - w/2 : i + w/2) = 1$$

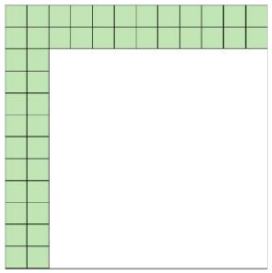


(b) Window attention

c. Global Attention

Global Attention

- A set of global tokens (g) attends to all parts of the sequence. Attend to all other tokens, and all other tokens attend them.
- e.g., the global attention mechanism with g = 1 and block size 2.



(c) Global Attention

Attention Computation

- When it comes to the computation of this attention score by simply multiplying arbitrary pairs of key and query vectors, it usually requires the use of the gather operation, which turns out to be inefficient.
- Upon examination of the global attention and window attention, it was found that these attention scores can be calculated without using a gather operation.
  - torch.gather(input, dim, index, out=None, sparse_grad=False): gathers values along an axis specified by dim (https://pytorch.org/docs/stable/generated/torch.gather.html)

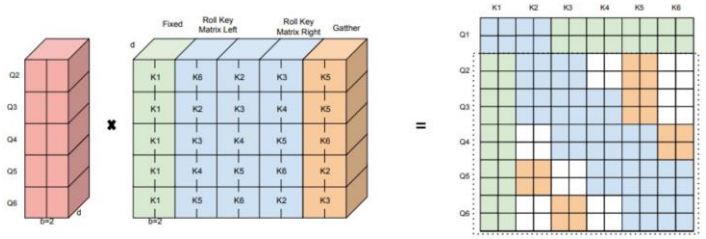
```
tensor([[ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9],
        [10, 11, 12, 13, 14, 15, 16, 17, 18, 19],
        [20, 21, 22, 23, 24, 25, 26, 27, 28, 29],
        [30, 31, 32, 33, 34, 35, 36, 37, 38, 39]])
```

⇒

```
tensor([[ 3],
        [17],
        [24],
        [31]])
```

BirBird Attention Computation

- The query tensor is obtained by **blocking** and **reshaping**, whereas the final key tensor is obtained by concatenating three transformations:
  - The first green columns (which corresponds to global attention) is fixed.
  - The center blue columns (corresponding to window local attention) are obtained by aptly rolling.
  - A computationally inefficient gather operation is supposed to be used for the last orange columns (which correspond to random attention).
- **Dense multiplication** between the query and key tensors effectively computes the sparse attention pattern (except for the first-row block that is calculated using direct multiplication).



Contribution & Limitations

Contributions

- **Sparse attention** mechanism **linear** in the number of tokens.
- SOTA on several NLP tasks upon release.

Limitations

- Switching to a **sparse attention** mechanism does incur a cost.