

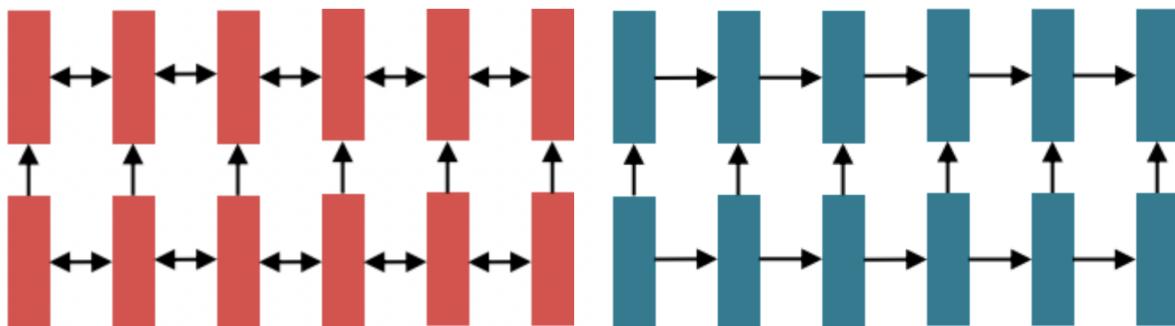
# Natural Language Processing with DeepLearning

## week 10

The course

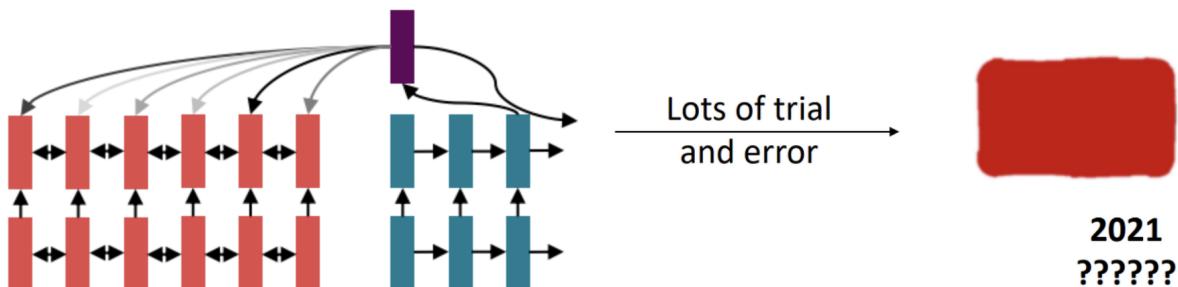
- From RNN to attention based NLP models
- Introducing the Transformer model
- Great results with Transformers
- Drawbacks and variants of Transformers

### 1. From RNN to attention based NLP models



1) 양방향 LSTM으로 문장 인코딩  
사용하여 생성

2) 출력을 시퀀스로 정의하고 LSTM



3) attention 이용하여 메모리에 유연하게 접근

<RNNs의 문제>

#### 1. 선형 상호작용 거리 (Linear interaction distance)

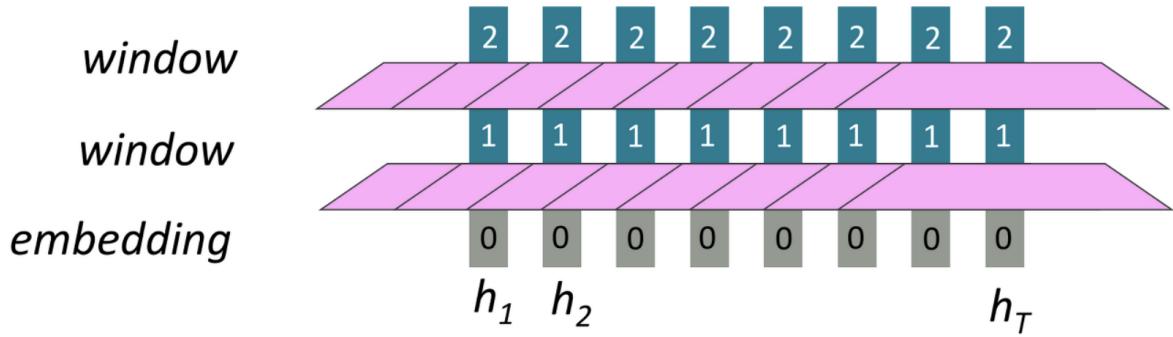
: RNNs은 "왼쪽에서 오른쪽으로" 전개되며 이는 선형 인접성을 인코딩함 -> 그러나 문제는 RNNs가 거리가 떨어진 단어간 상호작용하려면 O(시퀀스 길이) step이 필요-> 먼 거리의 존성은 기울기 소실 문제 때문에 제대로 학습하기 어려움. 단어의 선형 순서는 우리가 집어넣은 것으로 우리는 문장을 생각할 때 순서대로 보지 않음

#### 2. 병렬화의 부재

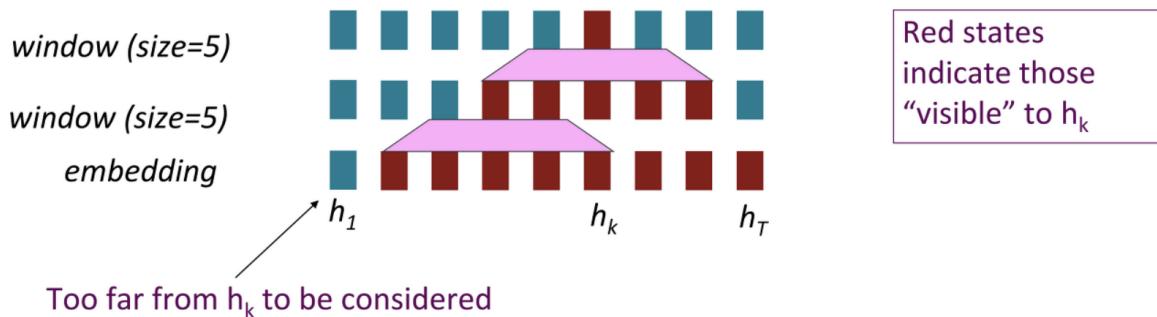
: 앞 혹은 뒤로의 흐름은 O(시퀀스 길이) 비병렬 연산을 수행해야 함-> GPU는 한 번에 많은 독립적인 계산을 수행할 수 있으나 과거의 RNN hidden states가 계산되기 전에는 미래의 RNN hidden states를 계산할 수 없음-> 이 때문에 매우 큰 데이터 세트에 적용할 수 없음

<If not recurrence, then what? How about word windows?>

: word window를 사용 시 시퀀스의 길이에 비례해서 비병렬 연산을 수행하지 않아도 됨

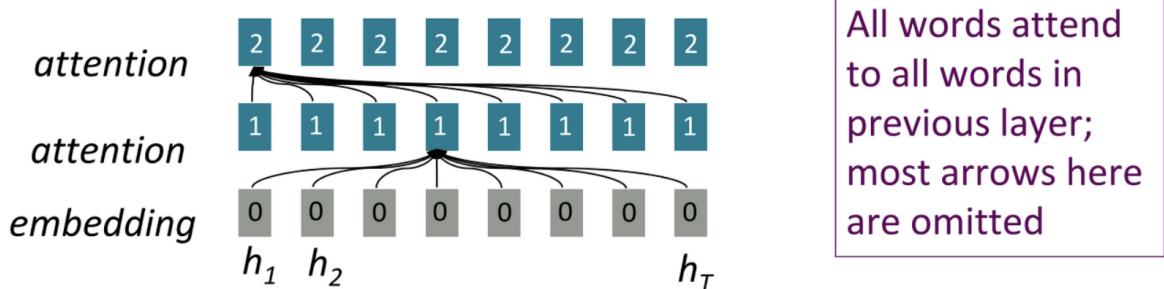


- word window는 주변 문맥을 모아서 볼 수 있기 때문에 층을 더 많이 쌓을 수록 이론상 더 많은 단어간 상호작용을 고려가능
- Maximum interaction distance = sequence length / window size
- 이론상 멀리까지 볼 수는 있지만 여전히 거리에 의한 제한이 존재하는 것은 사실임



<If not recurrence, then what? How about word attention?>

- 어텐션의 경우 각 단어의 representation에 query로 접근하여 values의 집합에서 정보를 얻음
- 이전에 우리는 decoder에서 encoder로 어텐션했지만 이번에는 단일 문장에 대해 생각! => 어텐션을 사용하면 문장 길이에 따른 비병렬화 문제는 해결
- Maximum interaction distance = O(1) 모든 단어가 모든 층에서 서로 상호작용



<self attention>

- 어텐션 연산을 위해서는 queries, keys, values가 필요

- queries  $q_1, q_2, \dots, q_T$ . 각 query는  $q_i \in \mathbb{R}^d$
  - keys  $k_1, k_2, \dots, k_T$ . 각 key는  $k_i \in \mathbb{R}^d$
  - values  $v_1, v_2, \dots, v_T$ . 각 value는  $v_i \in \mathbb{R}^d$
  - self-attention에서 queries, keys, values는 같은 source에서 생성된다.
- 만약 이전 층의 출력이  $x_1, x_2, \dots, x_T$  (단어 당 벡터 하나)라면,  $v_i = k_i = q_i = x_i$ 로 모두 같은 벡터이다.

$$e_{ij} = q_i^\top k_j \quad \alpha_{ij} = \frac{\exp(e_{ij})}{\sum_j \exp(e_{ij})} \quad \text{output}_i = \sum_j \alpha_{ij} v_j$$

Compute key-  
query affinities

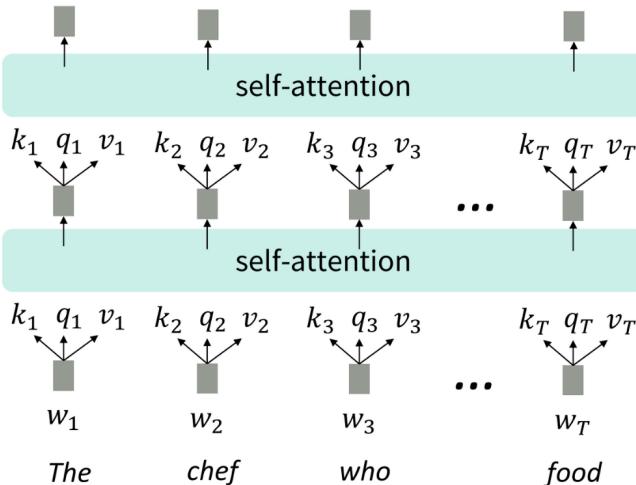
Compute attention  
weights from affinities

Compute outputs as  
weighted sum of values

Q) Fully connected layer와의 차이점은?

- Fully connected layer에서는 임의의 가중치를 설정해서 모델이 스스로 어떤 단어들 간의 상관성을 학습해야 하므로 학습이 느림.. 그러나 self-attention의 경우 모든 매개변수가 입력과 관계되어 있으며, 내적으로 연산을 수행하므로 층간에 모두 연결되지 않고 중요한 정보에 집중해서 층이 연결됨

<Self attention as an NLP building block>



Self-attention doesn't know the order of its inputs.

- LSTM을 쓰았던 것과 같이 그림에서 self-attention block을 쓴다. self-attention은 집합으로서 연산되기 때문에 순서에 대한 정보가 누락되어 있음, 다시 말하면 단어간 배열을 바꾸어도 똑같은 결과

만약 sequence index를 벡터로 표현한다면, 위치 벡터(position vector)  $p$ 는  $p_i \in \mathbb{R}^d, i \in \{1, 2, \dots, T\}$

이를 keys, queries, values에 반영하기 위해 간단하게 더하면

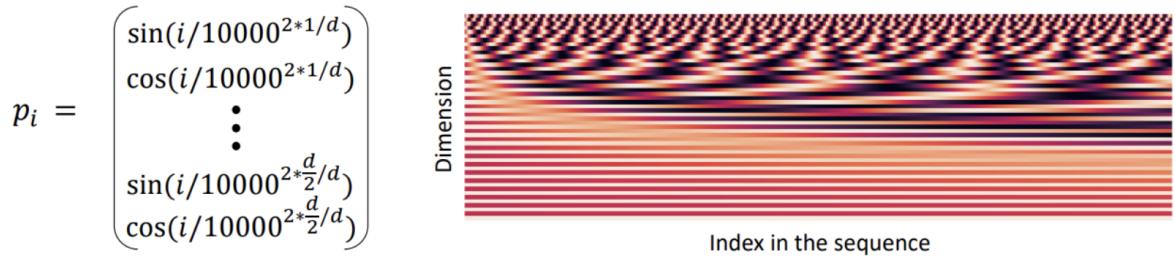
$$k_i = \tilde{k}_i + p_i$$

$$q_i = \tilde{q}_i + p_i$$

$$v_i = \tilde{v}_i + p_i$$

<Position representation vectors through sinusoids>

**Sinusoidal position representations:** 변화하는 주기에 따른 삼각함수를 연결



장점)

주기를 가진다는 뜻은 "절대 위치"는 중요하지 않다는 것을 뜻함

주기의 재시작을 이용하기 때문에 더 긴 시퀀스에도 적용가능함

단점)

학습이 불가능하며 실제로 긴 시퀀스에 적용이 잘 안됨

\*실제로 슬라이딩한 문장을 넣으면 생각보다 위치에 대한 의미 반영이 잘 안됨

**Learned absolute position representation:** 모든  $p$ 를 학습가능한 매개변수로 놓고 행렬

$p \in \mathbb{R}^{d \times T}$  를 학습하고 각  $p$ 가 해당 행렬의 열이 되도록 함

장점)

유연성: 각 위치는 데이터로 부터 학습된다

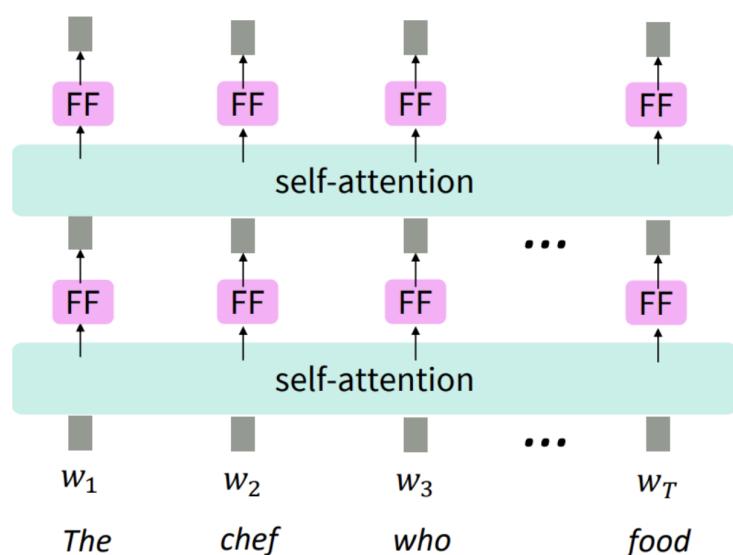
단점)

길이에 대한 확장이 불가능하다

\*학습한 문장보다 더 긴 문장을 넣을 수 없다.

<Adding nonlinearities in self-attention>

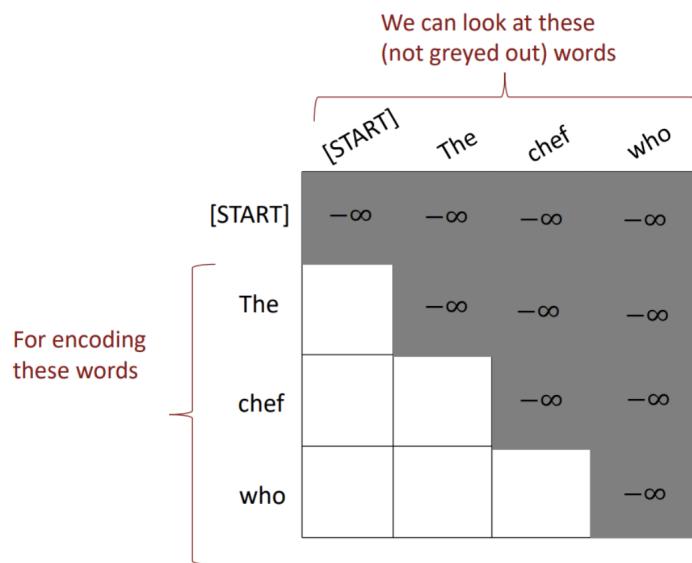
$$m_i = \text{MLP}(\text{output}_i) = W_2 \times \text{ReLU}(W_1 \times \text{output}_i + b_1) + b_2$$



<Masking the future in self-attention>

미래를 예측하는 기계번역이나 언어모델에서 미래를 보면 안됨 => 따라서 미래를 Masking  
기계 번역의 경우 encoder를 self-attention으로 설정하고 bi-directional LSTM을 decoder로  
사용한 이유는 모델이 실제 예측 단계에서 미래를 보지 않기 때문! 따라서 decoder로  
self-attention을 사용하기 위해서는

1. 매 timestep마다 과거의 정보만 가지도록 keys와 values를 변화시킴. (비효율적)
2. 병렬화를 위해서 미래 단어의 attention score를  $-\infty$ 로 mask out 해줌.



<정리>

### Barriers and solutions for Self-Attention as a building block

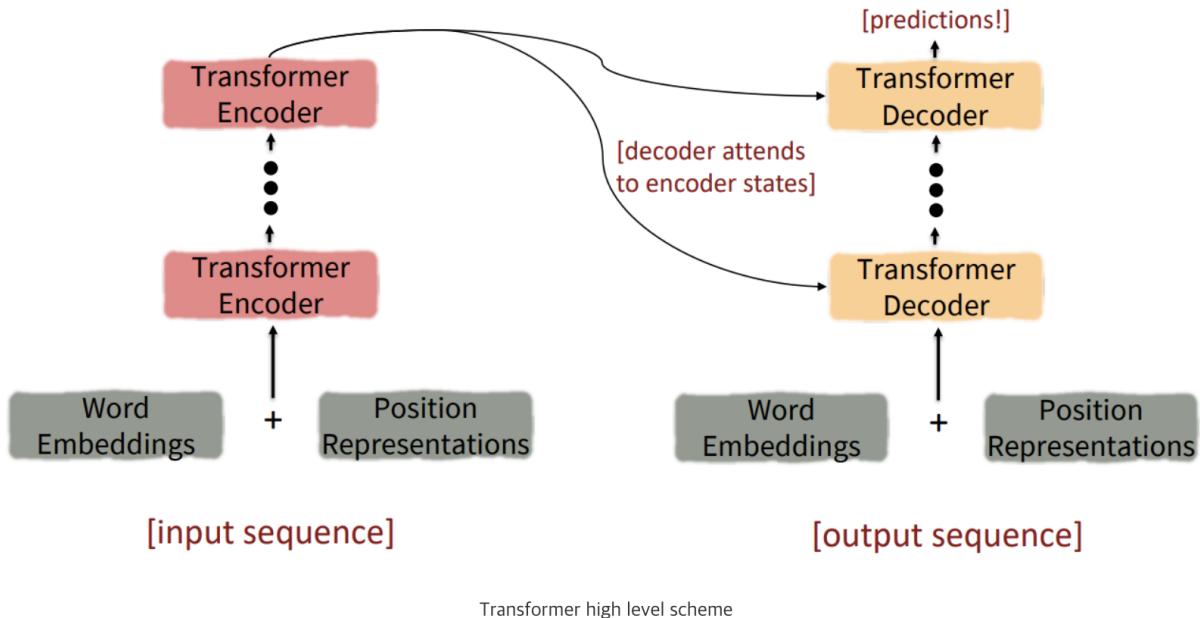
Barriers	Solutions
• Doesn't have an inherent notion of order!	→ • Add position representations to the inputs
• No nonlinearities for deep learning magic! It's all just weighted averages	→ • Easy fix: apply the same feedforward network to each self-attention output.
• Need to ensure we don't "look at the future" when predicting a sequence <ul style="list-style-type: none"> <li>• Like in machine translation</li> <li>• Or language modeling</li> </ul>	→ • Mask out the future by artificially setting attention weights to 0!

### Necessities for a self-attention building block:

- **Self-attention:**
  - the basis of the method.
- **Position representations:**
  - Specify the sequence order, since self-attention is an unordered function of its inputs.
- **Nonlinearities:**
  - At the output of the self-attention block
  - Frequently implemented as a simple feed-forward network.
- **Masking:**
  - In order to parallelize operations while not looking at the future.
  - Keeps information about the future from "leaking" to the past.
- That's it! But this is not the **Transformer** model we've been hearing about.

## 2. Introducing The Transformer model

<The transformer Encoder-Decoder>

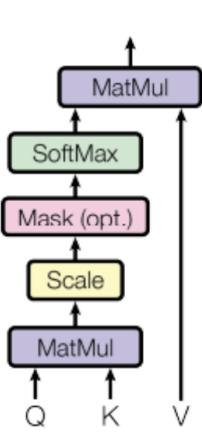


Transformer high level scheme

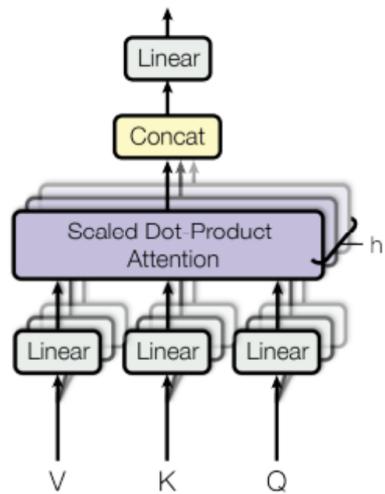
Transformer Encoder Block에서 다루지 않는 내용

1. key-query-value attention: 어떻게 한 단어에서  $k$ ,  $q$ ,  $v$  단어 임베딩을 수행하는가?
2. Multi-headed attention: 하나의 층에서 여러 곳으로 입력
3. 훈련에 도움되는 trick: 해당 trick들은 모델이 하는 일을 향상시키지는 않지만 훈련 과정을 향상시킨다.
  1. Residual connections
  2. Layer normalization
  3. Scaling the dot product

Scaled Dot-Product Attention



Multi-Head Attention



(left) Scaled Dot-Product Attention. (right) Multi-Head Attention consists of several attention layers running in parallel.

### <Transformer Encoder : Key-Query-Value Attention 연산>

We saw that self-attention is when keys, queries, and values come from the same source. The Transformer does this in a particular way:

- Let  $x_1, \dots, x_T$  be input vectors to the Transformer encoder;  $x_i \in \mathbb{R}^d$

Then keys, queries, values are:

- $k_i = Kx_i$ , where  $K \in \mathbb{R}^{d \times d}$  is the key matrix.
- $q_i = Qx_i$ , where  $Q \in \mathbb{R}^{d \times d}$  is the query matrix.
- $v_i = Vx_i$ , where  $V \in \mathbb{R}^{d \times d}$  is the value matrix.

These matrices allow *different aspects* of the  $x$  vectors to be used/emphasized in each of the three roles.

- Let's look at how key-query-value attention is computed, in matrices.
  - Let  $X = [x_1; \dots; x_T] \in \mathbb{R}^{T \times d}$  be the concatenation of input vectors.
  - First, note that  $XK \in \mathbb{R}^{T \times d}$ ,  $XQ \in \mathbb{R}^{T \times d}$ ,  $XV \in \mathbb{R}^{T \times d}$ .
  - The output is defined as output =  $\text{softmax}(XQ(XK)^\top) \times XV$ .

First, take the query-key dot products in one matrix multiplication:  $XQ(XK)^\top$

$$\begin{array}{ccc} XQ & \quad & XQK^\top X^\top \\ & K^\top X^\top & = \\ & & \text{All pairs of attention scores!} \end{array} \in \mathbb{R}^{T \times T}$$

Next, softmax, and compute the weighted average with another matrix multiplication.

$$\text{softmax} \left( \begin{array}{c} XQK^\top X^\top \end{array} \right) \times XV = \text{output} \in \mathbb{R}^{T \times d}$$

### <The Transformer Encoder: Multi-headed attention>

만약 문장 안에서 여러 곳을 한번에 집중하고 싶다면?

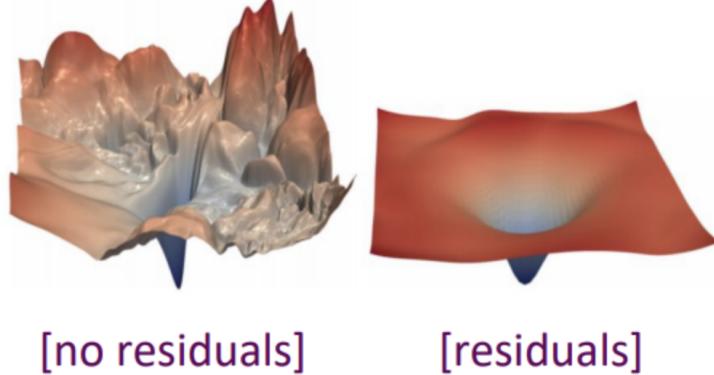
- What if we want to look in multiple places in the sentence at once?
  - For word  $i$ , self-attention “looks” where  $x_i^\top Q^\top K x_j$  is high, but maybe we want to focus on different  $j$  for different reasons?
- We'll define **multiple attention “heads”** through multiple Q,K,V matrices
- Let,  $Q_\ell, K_\ell, V_\ell \in \mathbb{R}^{d \times \frac{d}{h}}$ , where  $h$  is the number of attention heads, and  $\ell$  ranges from 1 to  $h$ .



### <The Transformer Encoder: Residual connections>

Residual connections는 모델의 학습을 도와주는 trick

$$X^{(i)} = X^{(i-1)} + \text{Layer}(X^{(i-1)})$$



이 기법을 이용하면 신경망이 깊어질 때 발생하는 기울기 소실 문제를 완화할 수 있음.  
Loss landscape 그림을 보면 residual connection을 적용하면 훨씬 표면이 부드러워지는 것을 확인할 수 있음=>. 따라서 학습에서 모델이 global minimum을 찾아가는 것이 훨씬 용이

#### <The Transformer Encoder: Layer normalization>

Layer normalization은 학습을 빠르게 해주는 trick

Idea: 은닉 벡터의 불필요한 정보 변동을 표준화를 통해 제거

$$\text{output} = \frac{x - \mu}{\sigma + \epsilon} * \gamma + \beta$$

Batch normalization과의 차이점)

Batch normalization의 경우 batch 크기에 의존하며, recurrent 기반 모델에 적용이 어렵다.  
기본적으로 입력이 sequence이기 때문에 batch normalization을 위해서는 매 timestep마다의 평균과 분산을 저장하여 적용해야하기 때문에 구현에 어려움이 존재 =>  
따라서 batch 크기에 의존하지 않으며 sequence 길이에도 영향을 받지 않는 방법이 layer normalization

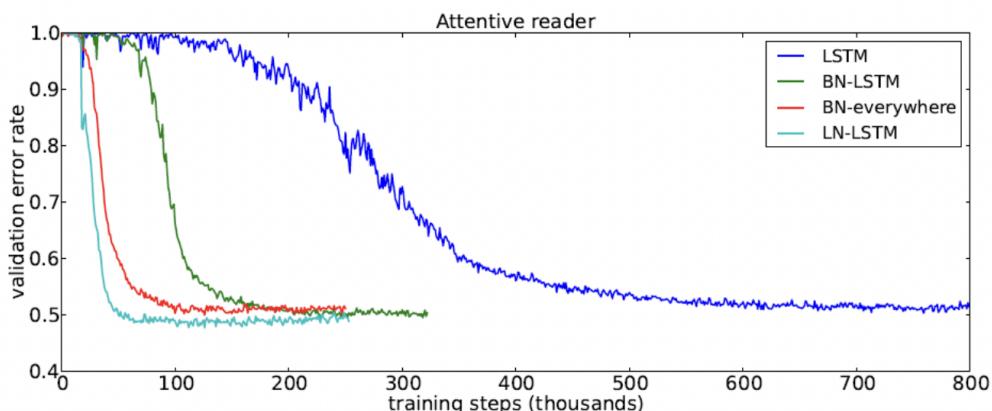


Figure 2: Validation curves for the attentive reader model. BN results are taken from [Cooijmans et al., 2016].

#### <The Transformer Encoder: Scaled Dot Product>

어텐션 score를 계산할 때 dot product 연산에서 차원 d가 증가함에 따라 그 값은 점점 커진다. score 값이 커지면 softmax 값의 일부 값이 굉장히 커지게 되고 낮은 확률을 지니고 있던 값들도 그 값이 너무 작아져 기울기 값이 0으로 수렴

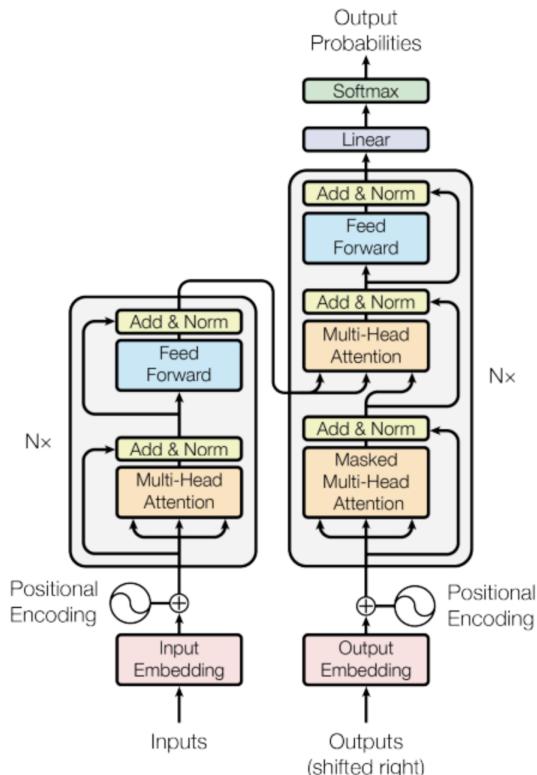
$$\text{output}_l = \text{softmax}(XQ_lK_l^TX^T) * XV_l, \text{output}_l \in \mathbb{R}^{\frac{d}{h}}$$

따라서 위의 self-attention 함수에서 attention score를  $\sqrt{d/h}$  해주어 차원 의존도를 없애준다.

$$\text{output}_l = \text{softmax}\left(\frac{XQ_lK_l^TX^T}{\sqrt{d/h}}\right) * XV_l, \text{output}_l \in \mathbb{R}^{\frac{d}{h}}$$

\*q와 k는 평균이 0 분산이 1이므로,  $q \cdot k = \sum_{i=1}^{d_k} q_i k_i$ 의 평균은 0 그리고 분산은  $d_k$ 가 된다.

### <The Transformer Encoder-Decoder>



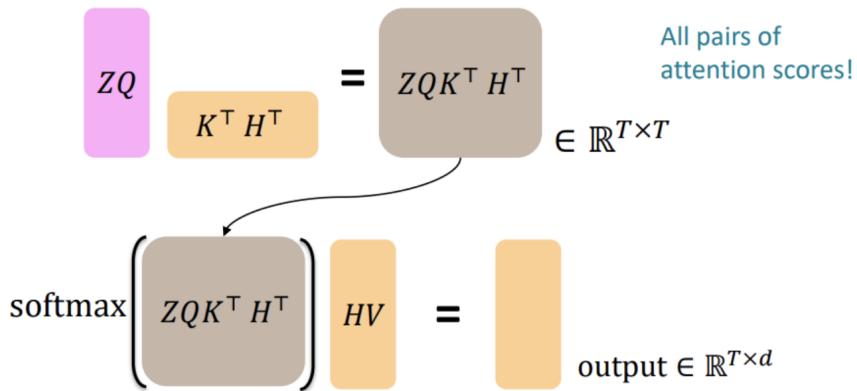
- 다시 모델 구조로 보면 매 층마다 residual과 layer normalization을 수행하며 feed forward는 attention 연산이 완료된 후에 적용함 이 때 encoder의 경우 단순 multi-head attion을 적용하고 decoder의 경우 미래를 추론하는 작업이기 때문에 미래에 대해서는 masking 된 attention을 적용!. 이후 출력값을 encoder의 출력값과 합쳐서 attention 연산을 수행하는데 소위 cross-attention이라 하는 이 부분에 대해 다루기!
- \*위의 그림에서 Nx로 되어 있는 부분을 통해 알 수 있듯이 해당 block을 여러 번 쌓을 수 있음 이 때 encoder의 경우 최종 출력값을 이용해 cross-attention에 넣어 주는데 그 이유는 각각 attention 출력을 decoder에 넣어주려면 우선 숫자가 같아야 구현이 용이하기 때문

### <The Transformer Decoder: Cross-attention (details)>

지금까지 우리는 self-attention에서 keys, querys, values를 같은 모두 소스로부터 가져오는 것 확인함. Decoder에서는 지난주 RNN어텐션에서와 유사한 작업을 수행

- Transformer encoder의 출력 벡터를  $h_1, \dots, h_T$  라고 놓자. 이 때  $h_i \in \mathbb{R}^d$
- Transformer decoder의 입력 벡터를  $z_1, \dots, z_T$  라고 놓자. 이 때  $z_i \in \mathbb{R}^d$   
(\* 입력 벡터: decoder의 입력층에 가까운 multi-head attention의 출력값)
- Keys와 values는 (메모리와 같이) encoder로부터 가져온다  
 $: k_i = Kh_i, v_i = Vh_i$
- Queries는 decoder로부터 가져온다.  
 $: q_i = Qz_i$
- Keys, queries, values가 정의되었으므로 cross-attention 연산을 수행해보자

- Encoder 벡터를 연결하여  $H = [h_1, \dots, h_T] \in \mathbb{R}^{T \times d}$ 라 하자.
- Decoder 벡터를 연결하여  $Z = [z_1, \dots, z_T] \in \mathbb{R}^{T \times d}$ 라 하자.  
\*연산의 편의성을 위해 같은 H와 Z를 같은 T차원으로 정의
- Cross-attention 출력:  $\text{output} = \text{softmax}(ZQ(HK)^T) \times HV$



### 3. Great results with Transformers

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	<b>41.29</b>	$7.7 \cdot 10^{19}$	$1.2 \cdot 10^{21}$
Transformer (base model)	27.3	38.1	<b><math>3.3 \cdot 10^{18}</math></b>	
Transformer (big)	<b>28.4</b>	<b>41.8</b>	$2.3 \cdot 10^{19}$	

Model	Test perplexity	ROUGE-L
seq2seq-attention, $L = 500$	5.04952	12.7
Transformer-ED, $L = 500$	2.46645	34.2
Transformer-D, $L = 4000$	2.22216	33.6
Transformer-DMCA, no MoE-layer, $L = 11000$	2.05159	36.2
Transformer-DMCA, MoE-128, $L = 11000$	1.92871	37.9
Transformer-DMCA, MoE-256, $L = 7500$	1.90325	38.8

On this popular aggregate benchmark, for example:



All top models are Transformer (and pretraining)-based.

Rank Name	Model	URL	Score
1	DeBERTa Team - Microsoft	DeBERTa / TuringNLv4	90.8
2	HFL fFLYTEK	MacALBERT + DLM	90.7
3	Alibaba DAMO NLP	StructBERT + TAPT	90.6
4	PING-AN Omni-Sinic	ALBERT + DAAF + NAS	90.6
5	ERNIE Team - Baidu	ERNIE	90.4
6	T5 Team - Google	T5	90.3

## 4. Drawbacks and variants of Transformers

- Self-attention에서의 이차함수적 연산량
  - 모든 짹의 상호작용을 연산한다는 것은 문장의 길이에 이차함수적으로 연산량이 증가한다는 것을 의미한다
  - 재귀 모델에서는 선형적으로 증가
- Position representations
  - 간단하게 절대적인 색인을 사용하는 것이 최선의 방법인가?
  - Relative linear position attention [Shaw et al., 2018]
  - Dependency syntax-based position [Wang et al., 2019]

<Quadratic computation as a function of sequence length>

- One of the benefits of self-attention over recurrence was that it's highly parallelizable.
- However, its total number of operations grows as  $O(T^2 d)$ , where  $T$  is the sequence length, and  $d$  is the dimensionality.

$$XQ \quad K^\top X^\top = XQK^\top X^\top \in \mathbb{R}^{T \times T}$$

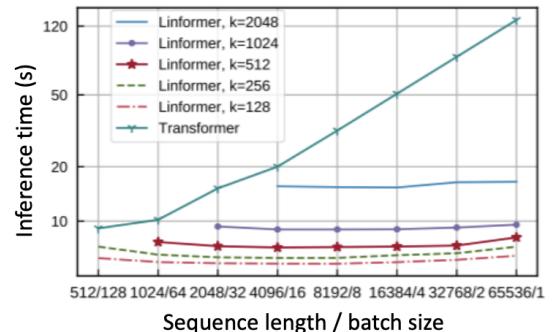
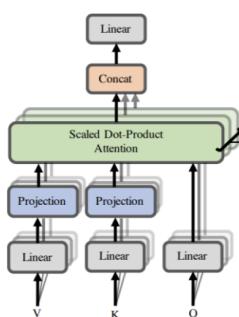
Need to compute all pairs of interactions!  
 $O(T^2 d)$

- Think of  $d$  as around 1,000.
  - So, for a single (shortish) sentence,  $T \leq 30; T^2 \leq 900$ .
  - In practice, we set a bound like  $T = 512$ .
  - But what if we'd like  $T \geq 10,000$ ? For example, to work on long documents?

### 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, Linformer [Wang et al., 2020]

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



## 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.**

