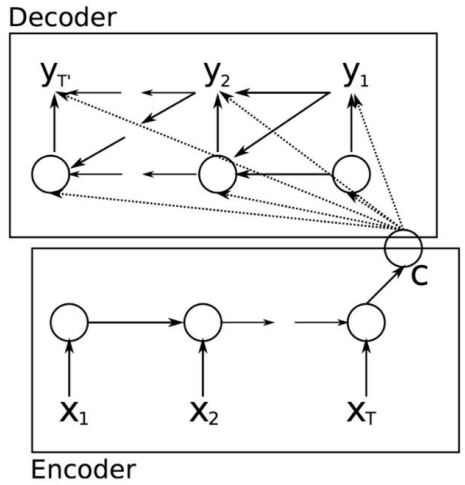
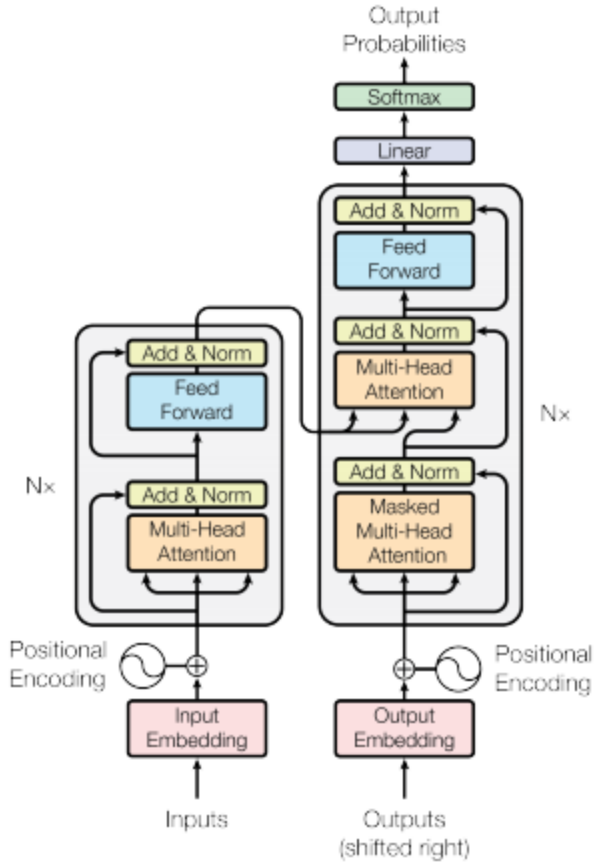
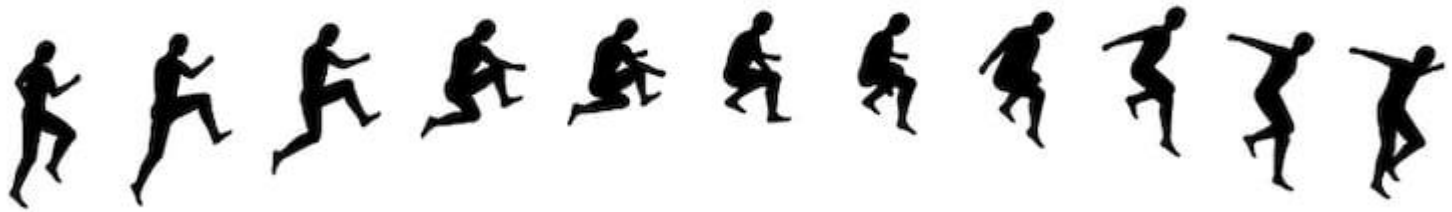
Seq2Seq

Recurrent에서 Transduction 모델까지
Attention을 중심으로





2020.04.07 화요일
발표자: MyungHoon-Jin
https://www.github.com/jinmang2



✓ Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation

- Seq2Seq 및 GRU 소개

✓ Neural Machine Translation by Jointly Learning to Align and Translate

- Attention Mechanism 소개

✓ A Structured Self-Attentive Sentence Embedding

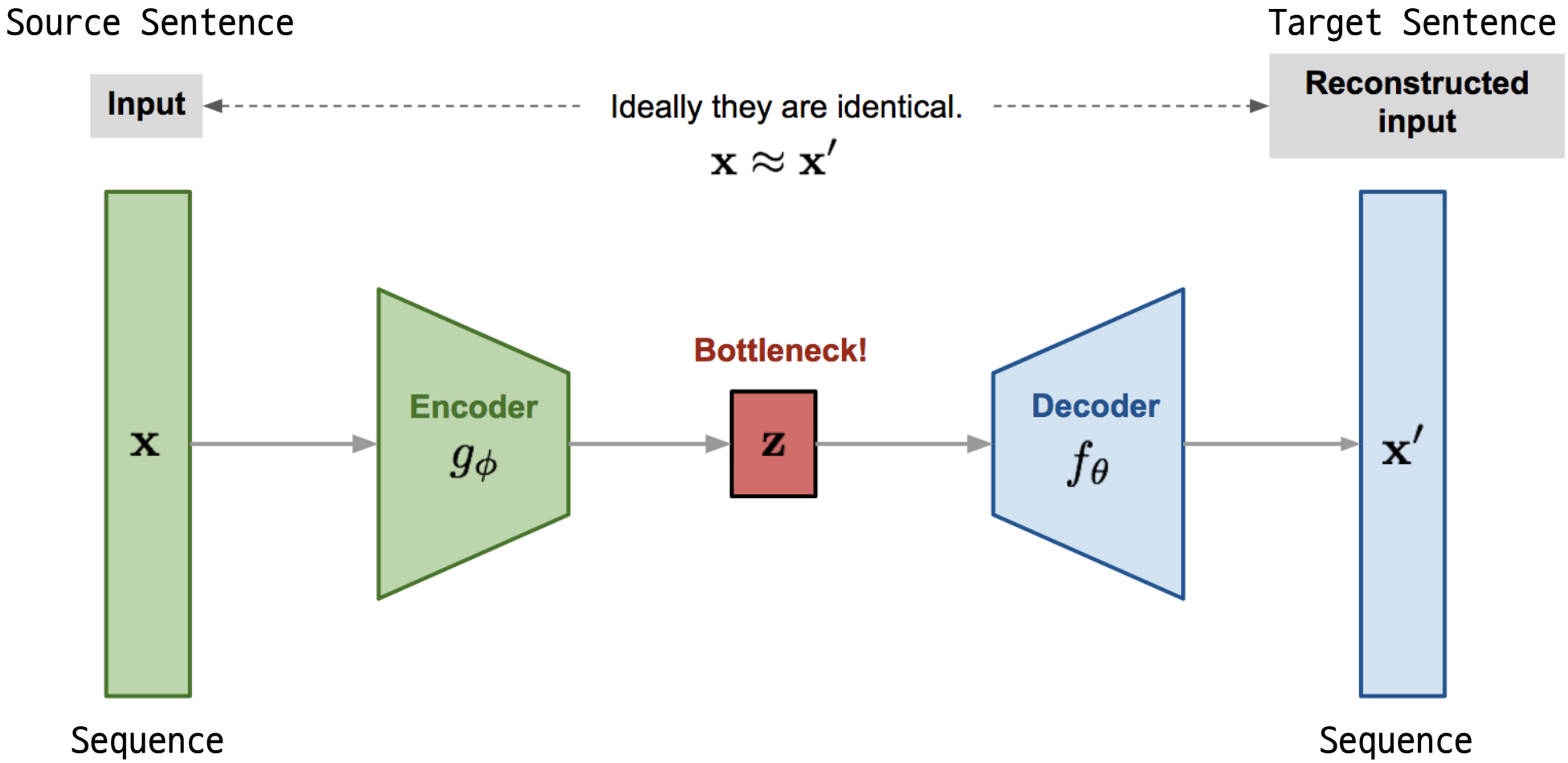
- Self-Attention Mechanism 소개

✓ Attention Is All You Need

- 1st Sequential Transduction Model

4 주간의 험난한 시간들…

간단히 요약!


Source Sentence
Target Sentence
Input
Ideally they are identical.
$\mathbf{x} \approx \mathbf{x}'$
Reconstructed input
$\mathbf{x}$
Encoder
g_ϕ
Bottleneck!
$\mathbf{z}$
Decoder
f_θ
$\mathbf{x}'$
Sequence
Sequence

1 RNN (Recurrent Neural Network)

RNN은 h, y, $x = (x_1, x_2, \dots, x_T)$로 이루어진 neural network

- h: hidden state
- y: optional output (one or many)
- x: variable-length sequence

각 time step t마다 x_t와 비선형 활성화 함수 f를 사용하여

$$h_{<t>} = f(h_{<t-1>}, x_t) \cdots (1)$$

로 업데이트 (f는 `logistic sigmoid function` 혹은 `LSTM`)

RNN은 다음 symbol을 예측하게 학습, 전체 sequence의 확률 분포를 배운다.
이 경우, 각 timestep t에서 output은 아래와 같은 조건부 확률

$$p(x_t|x_{t-1}, \dots, x_1)$$

예를 들어 `multinomial distribution`(1-of-K coding)은 `softmax` 함수를 사용해 출력 가능

$$p(x_{t,j} = 1|x_{t-1}, \dots, x_1) = \frac{exp(w_j h_{<t>})}{\sum_{j'=1}^{K} exp(w'_j h_{<t>})} \cdots (2)$$

$$where\ j = [1, K] : \text{possible symbols}\ \&\ w_j : \text{rows of a Weight Matrix}\ W$$

위 확률을 결합하여 sequence x의 확률을 계산 가능

$$p(x) = \prod_{t=1}^{T} p(x_t|x_{t-1}, \dots, x_1) \cdots (3)$$

위와 같이 분포를 학습하여 각 time step별 symbol을 반복적으로 추출, 새로운 sequence를 생성

2 Seq2Seq

확률적인 관점에서 seq2seq은 주어진 sequence에 대해 target sequence가 등장할 아래 조건부 확률을 학습

$$p(y_1, \dots, y_T|x_1, \dots, x_{T'})$$

$$note\ that: \ T \text{ is not always equal to } T'$$

- ***Encoder***
  Encoder는 `RNN`으로 input sequence x의 symbol을 sequentially하게 읽음
  각 symbol을 불러올 때마다 `RNN의` hidden state h는 $Eq(1)$과 같이 updated
  sequence를 전부 읽은 후의 `RNN의` hidden state는 전체 input sequence의 요약 c
- ***Decoder***
  제안된 모델의 Decoder는 hidden state h가 주어졌을 때 다음 symbol y_t를 예측하여 output sequence를 생성하는 `RNN`
  그러나 Encoder의 `RNN`과 다르게 y_t와 h는 이전 심볼 y_{t-1}과 input sequence의 요약 c의 영향을 받음
  때문에 각 time step t에서 decoder의 hidden state는 아래와 같이 update

  $$h_{<t>} = f(h_{<t-1>}, y_{t-1}, c)$$

  유사하게 다음 symbol의 조건부 확률을 아래와 같이 계산

  $$p(y_t|y_{t-1}, y_{t-2}, \dots, y_1, c) = g(h_{<t>}, y_{t-1}, c)$$

- ***Objective***
  제안된 seq2seq의 두 RNN은 아래 조건부 로그우도함수를 최대로 만들며 **동시에** 학습

  $$\max_{\theta} \frac{1}{N} \sum_{n=1}^{N} \log p\theta(y_n|x_n) \cdots (4)$$

  $$where\ \theta : \text{set of the model parameters}\ \&\ (x_n, y_n) : \text{(input seq, output seq) pair}$$

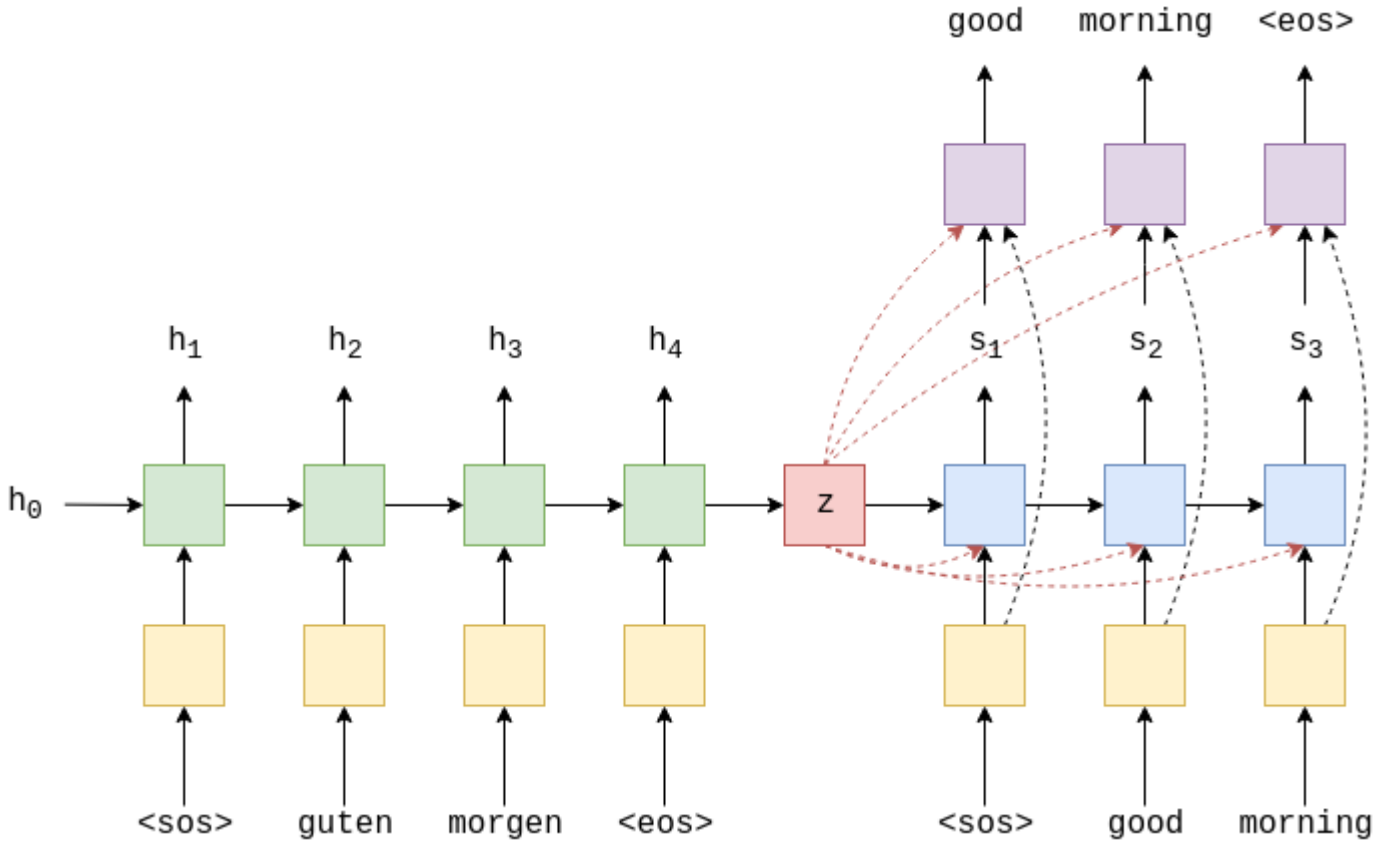
  input seq에서 output seq을 출력하는 모든 연산 과정은 모두 **미분 가능(differentiable)** 하기 때문에 model parameter를 추정하는데 `gradient-based algorithm`을 사용

Usage

`seq2seq`을 학습시킨 후에 모델을 아래 두 가지 방법으로 활용

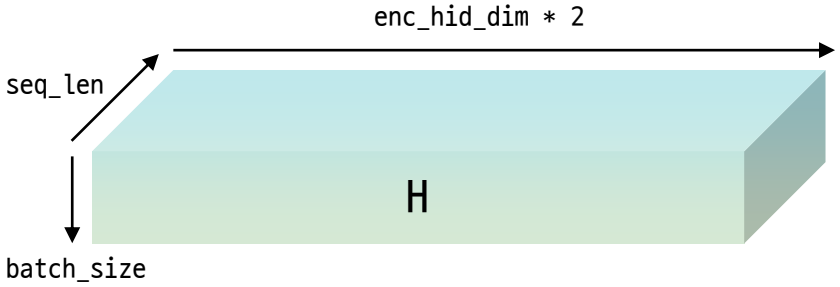
- input sequence가 주어졌을 때, 모델을 활용하여 target sequence를 생성
- $Eq(3)$과 $Eq(4)$의 확률값 $p\theta(y|x)$로 input과 output sequence pair에 score 부여

✓ Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation

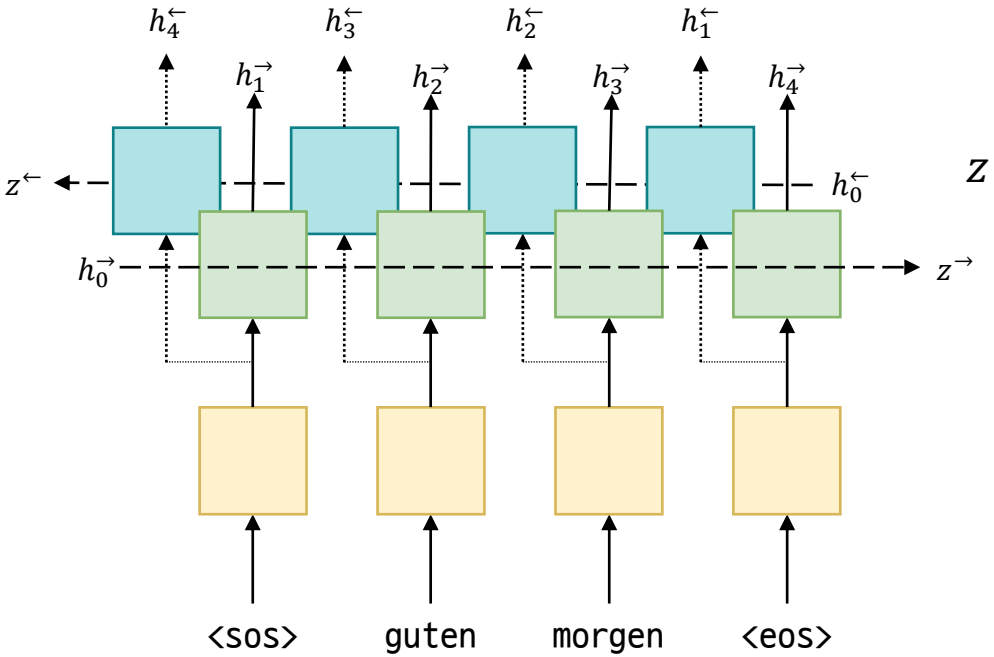


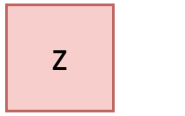
https://github.com/bentrevett/pytorch-seq2seq/blob/master/2%20-%20Learning%20Phrase%20Representations%20using%20RNN%20Encoder-Decoder%20for%20Statistical%20Machine%20Translation.ipynb

✓ Neural Machine Translation by Jointly Learning to Align and Translate

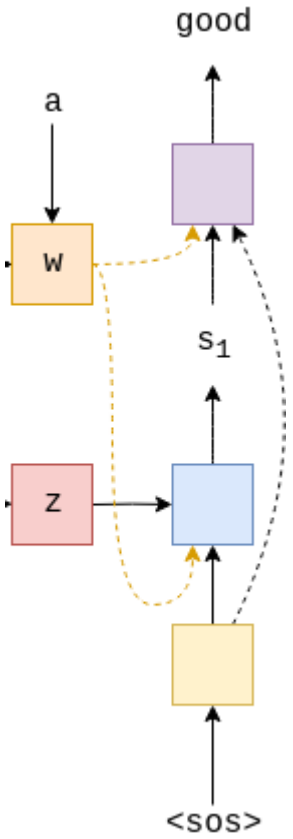


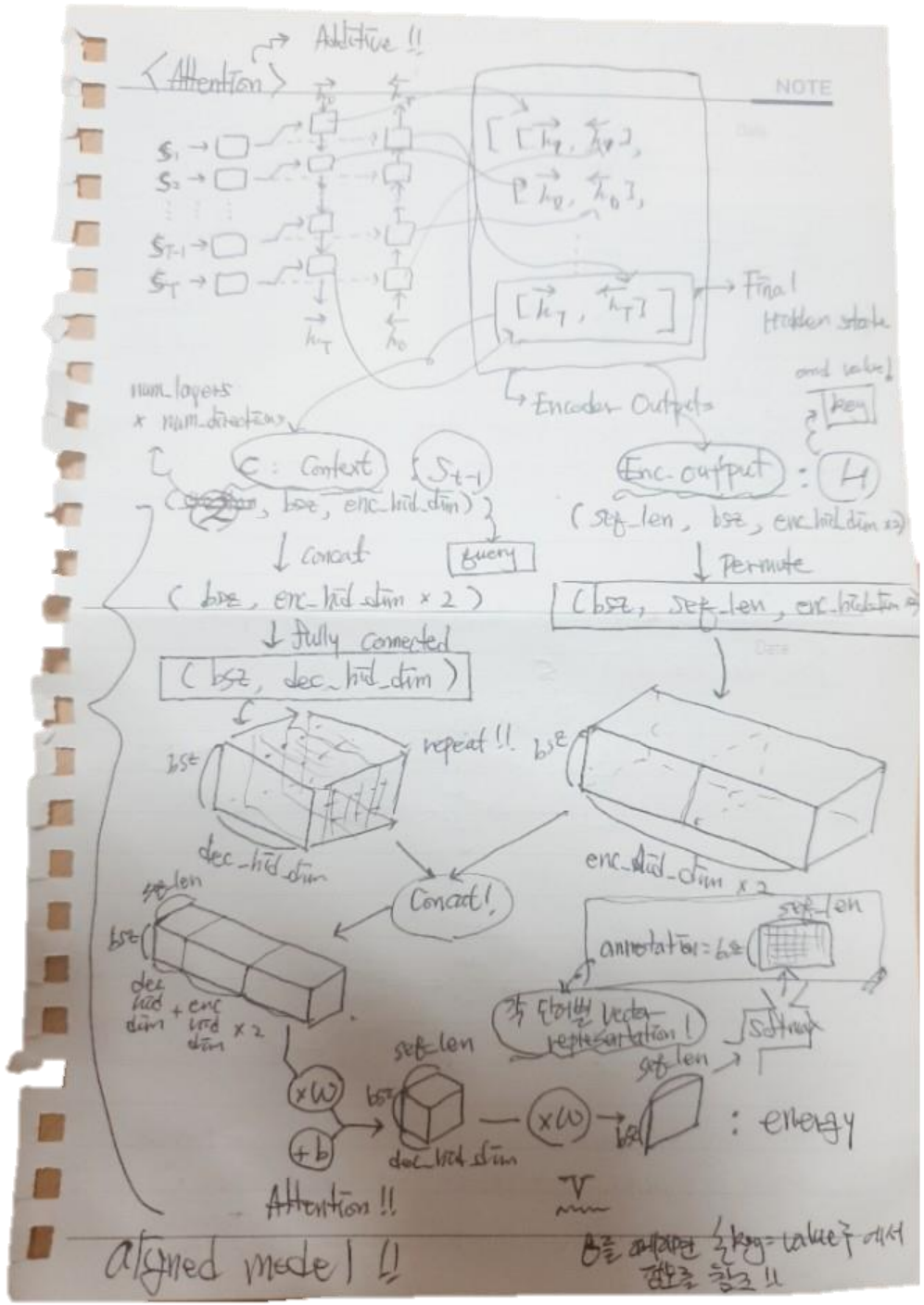
$$H = (h_1, h_2, \ldots, h_T) \quad where \;\; h_j = [\overrightarrow{h_j}, \overleftarrow{h_j}]$$

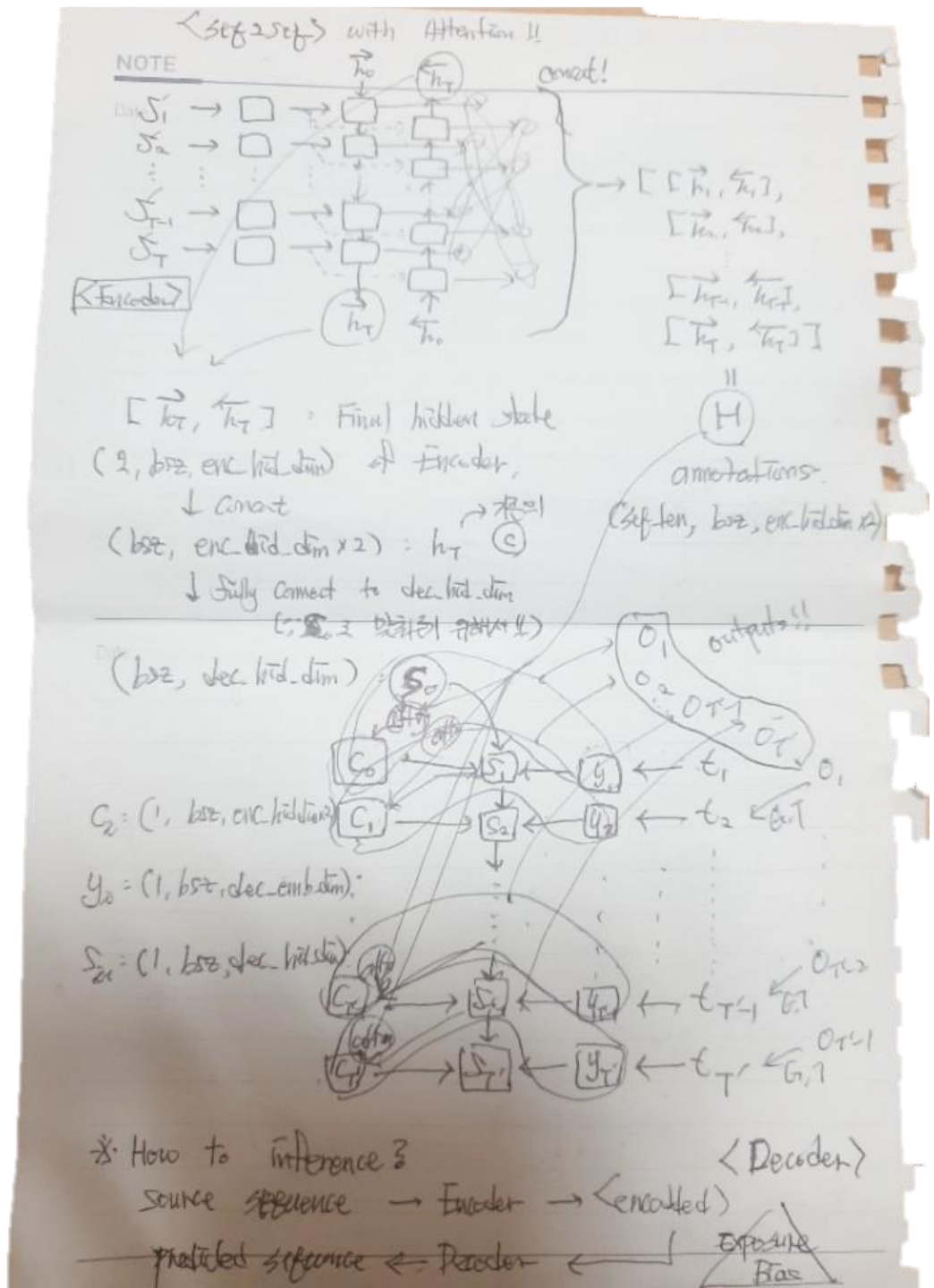




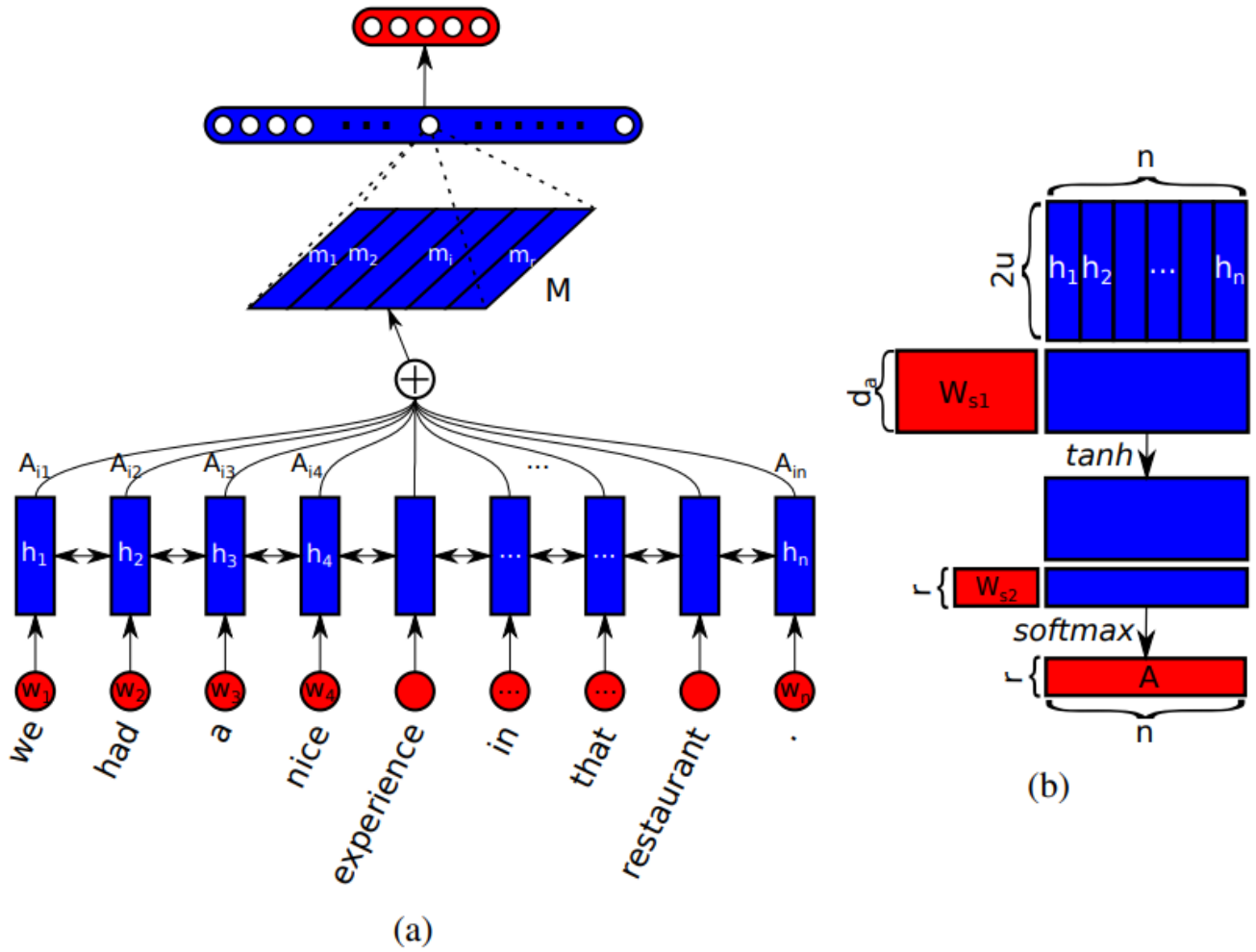
$$z = [z^{\rightarrow}, z^{\leftarrow}]$$




Attention
Additive !!
NOTE
Final Hidden state
Encoder Outputs
num_layers x num_directions
Context
Query
concat
fully connected
repeat !!
Concat!
Attention !!
: energy
aligned model !!


<seq2seq> with Attention !!
NOTE
concat!
Encoder
Final hidden state
annotations
outputs !!
Decoder
* How to inference?
source sequence → Encoder → <encoded>
predicted sequence ← Decoder ←

✓ A Structured Self-Attentive Sentence Embedding



✓ Attention Is All You Need

Abstract

현재의 우세한 `sequence transduction` 모델은 encoder와 decoder 구조를 사용하는 복잡한 RNN 내지 CNN에 기반을 둔다. 그 중 최상의 성능을 보인 모델은 Encoder와 Decoder를 `attention mechanism`을 사용하여 연결한 모델이다. 본 논문에서는 Recurrence/Convolution한 연산을 완전 배제한, 오직 attention mechanism에 중점을 둔 `Transformer`라는 단일 network architecture를 제안한다. 두 기계 번역 태스크에서 모델은 병렬적으로 우수한 성능을 보였고 학습에 시간도 오래 걸리지 않았다.
모델은 WMT 2014 English-German 번역 task에서 28.4의 BLEU score를 달성했으며 ensemble을 포함하여 현존하는 최상의 결과를 상회했다. WMT 2014 Engish-French 번역 task에서 모델은 단일 모델로 8개 GPU로 3.5일 정도 학습시켜 41.8의 BLEU 점수로 SOTA를 달성했다(아주 조금의 training costs 오차는 존재한다).

✓ Attention Is All You Need

1. Introduction

GRU와 LSTM같은 Recurrence Neural Network는 language modeling, machine translation과 같은 sequence modeling, transduction problem에서 SOTA를 달성했고 그 후 encoder-decoder 구조와 recurrent Language Model의 경계를 허무는데 많은 연구자들이 힘을 보탰다.

일반적으로 recurrent 모델은 입력 및 출력 시퀀스의 기호 위치를 따라 연산을 고려한다. 모델은 이전 hidden state h_{t-1}과 t 시점의 input의 함수 hidden state h_t를 생성한다. 본질적으로 이러한 순차적인 성질은(inherently sequential nature) 메모리 제약으로 인해 예제 간의 batch화가 제한되기 때문에 긴 sequence length를 가진 학습 예제의 병렬화를 배제한다. 최근 연구에서는 모델의 성능 개선은 차후로 미뤄두고 (1) factorization tricks, (2) conditional computation 등의 방법으로 계산 효율을 상당히 향상시켰지만, 그러나 여전히 sequential computation의 근본적인 제약은 남아있는 상태이다.

Attention Mechanism은 입력 또는 출력 시퀀스에서의 거리에 관계 없이 의존성을 모델링할 수 있도록 다양한 작업에서 매력적인 시퀀스 모델링 및 transduction 모델의 필수적인 부분이 되었다. 그러나 몇 가지 경우를 제외하고 attention mechanism은 RNN와 함께 사용된다.

본 연구에서 우리는 반복(recurrence)를 피하고(eschewing) 대신 input과 output 사이의 global dependencies를 도출하는(draw) attention mechanism에 전면적으로 집중한 단일 구조 **Transformer** 를 제안한다. Transformer는 병렬화를 가능케하고 8개의 P100 GPU로 12시간 정도 학습시킨 후에 새 경지의 SOTA를 달성했다.

✓ Attention Is All You Need

2. Background

sequential computation을 줄이는 목표는 병렬처리이다!

Extended Neural GPU, ByteNet, ConvS2S 등과 같이 만들기 위해서라고 이해해도 좋다. 이들은 `block` 으로 ConvNet을 사용, input과 output position을 병렬로 하여 hidden representation을 계산했다. 이러한 모델에서, 두 임의의 입력 또는 출력 위치의 신호를 연관시키는 데 필요한 연산의 수는 위치 사이의 거리에서 증가하며, ConvS2S의 경우 선형적으로 그리고 ByteNet의 경우 로그로 증가한다. 이는 얼마나 떨어져있는지에 대한 dependencies를 학습하기 어렵게 만든다. (왜?) **Transformer** 에서 *Multi Head Attention* 의 효과로 연산을 위와 같이 constant하게 줄였다. (`albeit at the cost of reduced effective resolution due to averaging attention-weighted positions`)

`Self-Attention` , `intra-attention` 은 sequence의 표현을 계산하기 위한 단일 sequence의 각 위치와 관련된 attention mechanism이다. self-attention은 reading comprehension, abstractive summarization, textual entailment, task-independent sentence representations 학습 등의 다양한 task에서 성공적으로 사용되었다.

`MemN2N` 은 sequence-aligned reccurence 대신 recurrent attention mechanism에 기반을 두고 있으며 simple-language QA 및 language modeling tasks에서 좋은 성능을 제시한다.

그러나! 우리가 아는 한 **Transformer** 는 sequence-aligned RNN 혹은 convolution을 제외하고 input과 output의 표현을 계산하기 위해 self-attention을 사용한 첫 transduction model이다.

3. Model Architecvture

대부분의 Neural Sequence Transduction Model들은 encoder-decoder의 구조를 가진다. Encoder는 $(x_1, \ldots, x_n)$의 symbol representation의 입력 문장을 $z = (z_1, \ldots, z_n)$의 continuous representation으로 매핑한다. 주어진 z로 decoder는 각 시점마다 한 symbol씩 생성하여 output sequence $(y_1, \ldots, y_n)$ sentence를 만든다. 각 스탭에서 모델은 다음 symbol을 생성할 때 이전에 생성된 symbol을 추가 input으로 활용하는 `auto-regressive`한 성질을 가진다.

Transformer는 encoder와 decoder 양쪽에 self-attention, point-wise 그리고 fc layer를 사용한 전반적인 구조를 따른다.

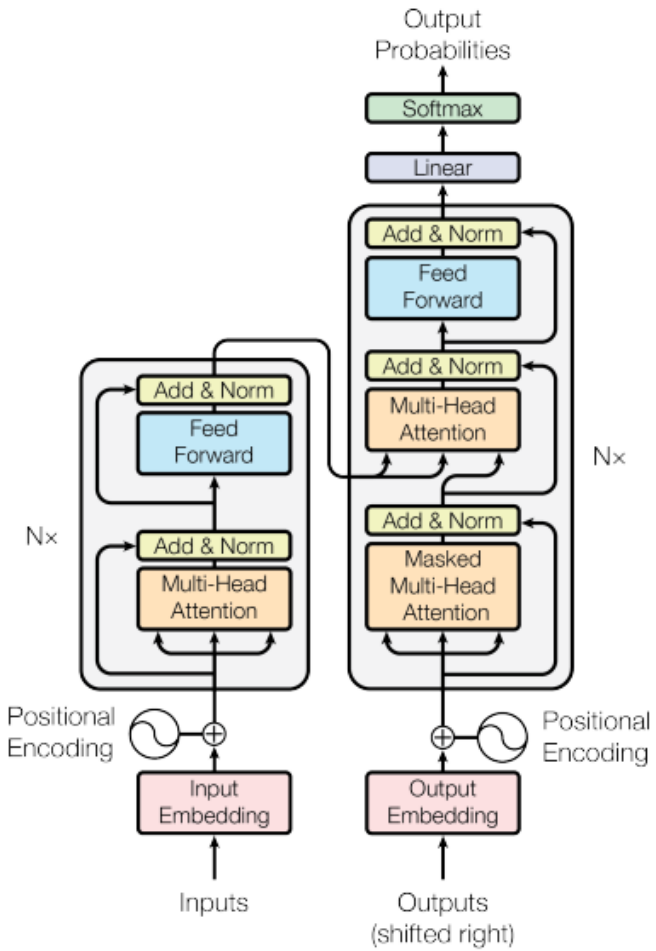


Figure 1: The Transformer - model architecture.

✓ Attention Is All You Need

3.1 Encoder and Decoder Stacks

Encoder: encoder는 6개의 같은 layer로 쌓여있다. 각 layer는 두 개의 sub-layer를 가진다. 첫 번째는 multi-head self-attention mechanism이며 두 번째는 간단한 position-wise fully connected feed-forward network이다. 여기에 layer normalization을 수행하고 각 sub-layer에 residual connection을 추가했다. 즉, 각 sub-layer의 output은 $\mathrm{LayerNorm}(x + \mathrm{Sublayer}(x))$이다. residual connection을 용이하게 수행하기 위해 모델의 모든 sub-layer와 embedding layers의 출력 차원은 $d_{model} = 512$로 설정했다.

Decoder: decoder도 6개의 같은 layer들로 쌓여있다. encoder layer의 두 sub-layer 사이에 encoder stack의 output에 대한 multi-head attention을 수행할 세 번째 sub-layer를 추가했다. encoder와 유사하게 각 sub-layer에 layer normalization과 residual connection을 수행했다. `We also modify the self-attention sub-layer in the decoder stack to prevent positions from attending to subsequent positions. This masking, combined with fact that the output embeddings are offset by one position, ensures that the predictions for position i can depend only on the known outputs at positions less than i.`

✓ Attention Is All You Need

3.2 Attention

Attention function은 `query` 와 `key-value` pairs의 set을 output을 mapping한다(`query` , `key` , `value` , `output` are all vectors). output은 value들의 가중합으로 계산되며 각 값에 할당된 가중치는 `query` 와 이에 연관된 `key` 의 `compatibility function` 으로 계산된다.

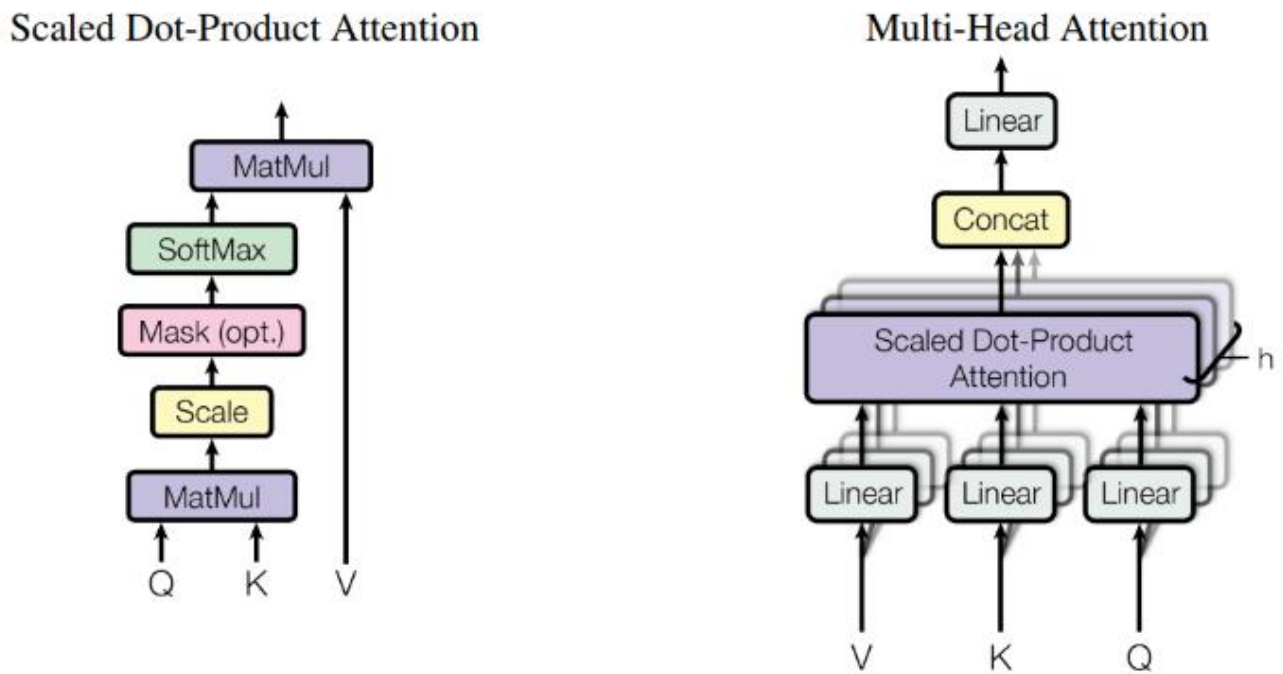


Figure 2: (left) Scaled Dot-Product Attention. (right) Multi-Head Attention consists of several attention layers running in parallel.

3.2.1 Scaled Dot-Product Attention

input은 d_k의 차원을 가진 query와 key, 그리고 d_v의 차원을 가지는 value들로 구성된다. 자, 모든 `key` 와 `query` 의 dot product(내적)을 계산하고 이를 $\sqrt{d_k}$로 나눈 다음, 각 `value` 의 가중치를 얻기위해 `softmax` 함수를 적용한다.

실험에서 모든 `query` 를 matrix Q로 동시에 packing하고 attention function을 계산했다. `key` 와 `value` 또한 행렬 K와 V로 pack된다. 이제 output 행렬을 아래와 같이 계산하자.

$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V \cdots (1)$$

가장 흔하게 쓰는 `attention functions` 은 `additive model` 과 `dot-product(multiplicative)` attention 이 두 가지이다. 전자는 Neural Machine Translation by Jointly Learning to Align and Translate에서 소개된 방식이고 `Dot-product attention` 이 우리의 알고리즘이다. 아, `scaling factor` $\frac{1}{\sqrt{d_k}}$가 추가됐긴 했다 하하. `Additive attention` 은 단일 hidden layer를 가진 feed-forward network를 사용하여 `compatibility function` 을 계산한다. 이 두 attention이 이론적으로 복잡한 것은 같지만 `Dot-product Attention` 은 더 빠르고 실험에서 더욱 메모리 효율적이다. 왜냐고? 이는 `highly optimized matrix multiplication code` 를 사용하여 구현되었기 때문이다 ㅎㅎ.

작은 d_k를 사용했을 때 두 mechanism의 성능은 유사했지만, 큰 값의 d_k에 대해선 `additive attention` 이 scaling factor를 적용하지 않은 `dot-product attention` 보다 우월했다. 우리는 d_k의 큰 값에 대해 내적 (dot product)의 크기가 커져서 softmax 함수가 gradient가 매우 작은 영역으로 푸시되는 것으로 의심합니다(즉, 학습이 잘 되지 않는다).

- 왜 내적 값이 커질까?
- q와 k의 요소가 서로 독립인 평균 0, 분산 1을 가지는 random variable이라 가정하자.
- 이 때 q와 k의 내적 $q \cdot k = \sum_{i=1}^{d_k} q_i k_i$는 평균 0 그리고 분산 d_k를 가진다.
- 때문에 d_k가 커질수록(차원이 커질수록) variation이 늘어나고 이에 내적 값이 커질 가능성이 커지며
- 이 값에 exponential을 씌워 softmax를 만들면 극단적인 값이 생성된다.

이에 대한 대책으로 우리는 내적 값을 $\frac{1}{\sqrt{d_k}}$를 곱해 scaling했다.

3.2.3 Applications of Attnetion in our Model

Transformer는 multi-head attention을 아래 세 방법으로 사용한다.

1. `Encoder-Decoder attention` layer에서 query는 이전 decoder layer에서, memory key와 value는 encoder output에서 온다. 이는 decoder의 모든 position이 입력 시퀀스의 모든 position에 집중할 수 있게한다. 이는 [38, 2, 9]와 같은 전형적인 seq2seq model의 encoder-decoder attention mechanism을 흉내낸다.
2. Encoder는 selt-attention layer를 포함한다. attention layer에서 모든 key, value, query는 같은 공간, encoder의 이전 layer의 output에서 나온다. encoder의 각 position은 encoder의 이전 layer의 모든 position에 집중한다.
3. 유사하게, decoder의 self-attention layer들은 decoder의 각 position이 해당 position까지, 혹은 이를 포함하는 모든 position에 대해 집중할 수 있게한다. 우리는 `auto-regressive` 성질을 보존하기 위해 decoder의 `leftward information` flow를 방지해야 한다. 이에 우리는 `scaled dot-product attention` 내부를 구현함에 있어 softmax의 input에 $-\infty$로 masking하여 `illegal`하게 연결되도록 한다.

3.3 Position-wise Feed-Forward Networks

attention의 sub-layers이외에 encoder와 decoder의 각 layer들은 각 position에 대해 분리적, 동일하게 적용되는 fully connected feed-forward network를 또 하나 가지고 있다. 두 linear transformation 사이에 ReLU activation을 낀 ffn은 아래와 같다.

$$\mathrm{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2 \cdots (2)$$

linear transformation은 서로 다른 위치에서 위와 같이 동일하게 실시하지만 layer에서 layer로 갈 때는 다른 paremeter를 사용한다. 이를 다르게 설명하자면 kernel size를 1로 둔 두 convolution을 생각하면 될 것 같다. input와 output 차원은 $d_{model} = 512$이며 inner-layer는 $d_{ff} = 2048$ 차원을 가지고 있다.

3.4 Embeddings and Softmax

다른 sequence transduction 모델과 유사하게 우리는 input, output token을 d_{model} 차원의 벡터로 바꾸는 embedding을 학습할 때 모델을 사용한다. 또한 다음 token의 등장 확률을 예측하기 위해 decoder의 output을 변환할 linear transformation과 softmax 함수를 사용한다. 우리의 모델에서 두 embedding layer와 pre-softmax transformation 사이의 두 weight matrix를 [30]과 유사하게 공유한다. embedding layer에서 우리는 두 weight에 $\sqrt{d_{model}}$을 곱해줬다.

3.5 Positional Encodeing

우리의 모델에는 recurrence, convolution 연산이 없기 때문에 model이 sequence 순서를 만들어내기 위해 우리는 sequence token들에 관련된 혹은 자체 position 정보를 주입해야 한다. 우리는 위 정보를 주입하기 위해 다른 주기를 사용한 sine과 cosine 함수를 사용했다.

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

pos는 position이며 i는 차원이다. 즉, 각 positional encoding의 차원은 `sinusoid`에 연관돼있다. 등비수열의 파장(`wavelengths from a geometric progression`)은 2π에서 $10000 \cdot 2\pi$까지 사용된다. 우리는 위 `sinusoid`함수가 고정된 offset k에 대해 PE_{pos+k}가 PE_{pos}의 linear function으로 표현될 수 있기 때문에 우리의 모델이 관련된 position에 쉽게 집중할 수 있게 할 것이라 생각했고 이를 사용했다.

또한 [9]의 positional embedding을 위의 방법과 함께 실험했고 두 version이 거의 같은 결과를 제공한다는 것을 밝혔다. 우리는 model이 학습 도중에 마주친 sequence보다 더 긴 sequence 길이로 추정(`extrapolate`)할 수 있을 거라 기대하고 `sinusoidal` version을 사용했다.

4. Why Self-Attention

자, 이번 section에서 우리는 self-attention layer를 recurrent, convolutional layer를 비교하자. recurrent, convolutional layer는 variable-length의 symbol representation $(x_1, \dots, x_n)$을 다른 같은 길이의 sequence $(z_1, \dots, x_n)$으로 mapping하는데 자주 사용되며 각 x_i, x_i는 $\mathbb{R}^d$의 vector이다. 우리의 self-attention은 아래 세 가지 `desiferata`에 영감을 받았다.

하나는 각 layer의 computational complexity, 둘 째로 계산의 양이 필요로 하는 순서열 계산이 최소가 되도록 병렬로 가능할 것.

마지막은 네트워크의 긴 길이의 dependencies사이의 path length이다. 많은 sequence transduction task에서 `long-range dependencies`를 학습하는 것은 key challenge였다. 이러한 dependencies를 학습시키는데 중요한 한 가지 요소는 네트워크에서 통과(traverse)해야 하는 전방 및 후방 신호의 길이이다. 입력과 출력 수열의 position의 조합 사이의 거리가 적을 수록 long-range dependencies를 학습히키기 쉽다는 연구 결과가 이미 존재한다[12]. 이에 우리는 각기 다른 layer로 구성된 network의 두 입력과 출력 position 사이의 최대 path 길이를 비교했다.

Table 1의 결과를 보면, recurrent layer는 $O(n)$의 수열 연산을 요구하지만 self-attention layer는 모든 position을 고정된 수열 연산을 사용하여 연결한다. `computational complexity`적 관점에서(`in terms of`), self-attention layer는 sequence 길이 n이 representation 차원 d보다 작을 경우 recurrent layer보다 빠르다 ($n < d$). 이는 `word-piece`[38], `byte-pair`[31] 표현과 같은 기계 번역에서 SOTA를 달성한 sentence representation 모델에서 흔히 등장하는 경우이다 (대부분의 경우라는 소리). 아주 긴 sequence를 포함하는 task의 성능을 올리기 위해 self-attention은 각 출력 position을 중심으로 한 입력 sequence에서 size r만큼의 이웃만 고려하도록 제한한다. 이는 최대 path 길이가 $O(n/r)$까지만 증가하도록 하며 우리는 향후 연구에서 이에 대해 연구하도록 하겠다.

kernel width $k < n$의 단일 convolutional layer은 모든 입출력 위치를 연결시킬 수 없다. 연속된 kernel로 $O(n/k)$의 conv layer로 쌓거나 $O(log_k(n))$의 `dilated convolutions`[18]로 수행해야 네트워크에서 두 포지션 사이의 최장 길이로 늘릴 수 있다. conv layer는 일반적으로 k 때문에 recurrent layer보다 cost가 많이 든다. 그러나 `Separable convolutions`[6]은 복잡도를 $O(k \cdot n \cdot d + n \cdot d^2)$로 줄였다. 그러나 $k = n$일 때 `Separable convolutions`의 복잡도는 우리 모델의 self-attention layer와 point-wise feed-forward layer의 조합과 같았다.

이러한 이점으로 self-attention은 더 해석적인 모델로써 탄생됐다. 우리는 모델의 attention distribution을 살펴봤고 표현했으며 appendix의 예시로 토의했다. 각기 attention head가 다른 task 수행을 학습했을 뿐만 아니라 문장 구조의 의미적/문법적인 관계까지 포착했음을 도출했다.

5.2 Hardware and Schedule

8개의 NVIDIA P100 GPUs로 학습, 논문의 hyperparameters를 하용하여 학습 시켰을 때 각 training step별 0.4초정도 걸렸다. 전체 100,000 step을 학습시키며 12시간이 걸렸다. big model을 학습시키는 데에는 step당 1.0초가 걸렸으며 총 300,000 step(3.5일)을 학습시켰다.

5.3 Optimizer

학습에 Adam[20]을 사용했으며 $\beta_1 = 0.9, \beta_2 = 0.98, \epsilon = 10^{-9}$로 설정했다. 학습에 아래 수식의 변하는 learning rate를 사용했다.

$$lrate = d_{model}^{-0.5} \cdot \min(step_num^{-0.5}, step_num \cdot warmup_steps^{-1.5}) \cdots (3)$$

학습 중 첫부터 $warmup_steps$까지 선형적으로 증가하며 step의 inverse square root값만큼 비율적으로 감소한다. 우리는 $warmup_steps$를 4,000으로 사용했다.

5.4 Regularization

3 가지의 regularization을 사용한다(근데 왜 2개지?).

Residual Dropout
각 sub-layer의 출력에 dropout[33]을 적용하고 이를 sub-layer 입력에 추가하여 정규화한다(맞나?). 추가로 encoder와 decoder stack 둘의 positional encodings과 embedding의 합에도 dropout을 적용했다. base model에서 $P_{drop} = 0.1$을 적용했다.

Label Smoothing 학습 중 [36]의 label smoothing을 $\epsilon_{ls} = 0.1$로 수행했다. 이는 모델의 perplexity를 증가시키며 모델이 모호함을 느끼게 만들지만 정확도와 BLEU score를 향상시켰다.

https://github.com/simonjisu/nsmc_study/blob/master/Notebooks/selfattn_3H_r5.ipynb

https://github.com/bentrevett/pytorch-seq2seq/blob/master/6%20-%20Attention%20is%20All%20You%20Need.ipynb

https://github.com/kh-kim/simple-nmt/blob/master/simple_nmt/transformer.py#L136

이 다음엔...

1. 오늘 발표 내용 깔끔히 정리
2. Transformer 입출력 내용 정리
3. Batch, Head 별로 weight 시각화하여 dropout, Layer Normalization 효과 확인하기
4. Inference를 어떻게 할 것인가 (Beam-Search)
5. 김기현님 구현체 vs 원 저자 구현체 vs cchyun님 구현체 비교 분석