Attention_Is_All_You_Need_team5

📌 **파트 분배**

- 강민정: 1. Introduction ~ 3.2.1 Scaled dot-product attention
- 이동주: 3.2.2.Multi-head attention~ 5.3 Optimizer
- 이유진: 5.4 Regularization ~ Conclusion

1. Introduction

- 시퀀스 모델링 : RNN, LSTM, GRU (Gated Recurrent Neural Networks)
  - 순환 언어 모델 및 인코더-디코더 구조 사용
- 순환 모델 (Recurrent model)
  - 입력과 출력 시퀀스의 위치에 따라 연산을 수행 : 이전 시점의 hidden state(t-1)와 현재 시점의 input(t)을 바탕으로 hidden state(t) 생성
  - 병렬화 불가 : 긴 시퀀스의 경우 계산 효율성 떨어짐

    → 순차적 연산의 한계
- Attention
  - 입력과 출력 시퀀스 간 거리와 상관없이 모델링 가능
- Transformer
  - 순차적 연산 대신 어텐션 메커니즘을 사용해 입력과 출력 시퀀스의 관계 파악
  - 병렬화 가능

2. Background

- Extended Neural GPU, ByteNet, ConvS2S
  - 순차적 연산을 줄이기 위해 CNN (Convolutional Nerual Networks) 사용

    → 입력과 출력 시퀀스의 거리에 비례해 연산 증가
  - 이러한 문제를 해결하기 위해, Transformer는 연산 횟수를 일정하게 유지
- Self-attention
  - 하나의 시퀀스 내에서 다른 위치 간 관계를 계산해 시퀀스의 전체 표현 생성
- End-to-end memory

    - 시퀀스의 정렬된 순환 대신 recurrent attention mechanism 활용
- Transformer
    - 시퀀스 모델 대신 self-attention만을 사용해 입력과 출력을 계산한 모델

3. Model Architecture

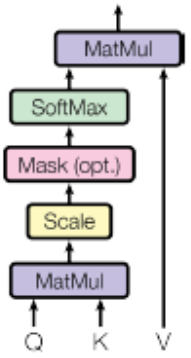
- Encoder-decoder structure
    - Encoder : 입력 시퀀스를 연속적인 벡터로 변환
    - Decoder : 출력 시퀀스 생성
- Transformer 모델은 인코더와 디코더에서 stacked self-attention, point wise fully connected layer 사용

3.1 Encoder and Decoder Stacks

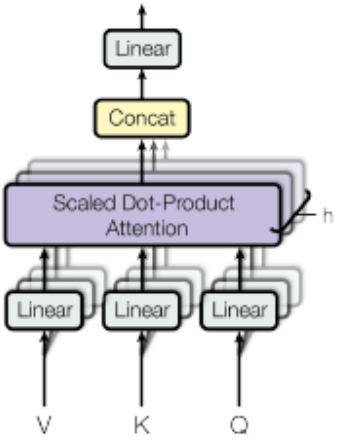
- Encoder
    - 6개의 동일한 레이어로 구성
    - 각 레이어는 2개의 sub-layer로 구성 (multi-head attention, feed-forward)
        - sub-layer는 잔차 연결과 레이어 정규화가 적용됨
- Decoder
    - 6개의 동일한 레이어로 구성
    - 각 레이어는 3개의 sub-layer로 구성 (masked multi-head attention, multi-head attention, feed-forward)
        - sub-layer는 잔차 연결과 정규화가 적용됨
        - masked multi-head attention은 이후 위치를 참조하지 않음

3.2 Attention

Scaled Dot-Product Attention



Multi-Head Attention



- Input : Query, Key-Value
- Output : 각 Query와 Key의 호환성 함수에 의해 할당된 가중치와 Value의 가중합

3.2.1 Scaled Dot-Product Attention

- Attention function : Additive attention, Dot-Product Attention

  → Additive attention : Dot-Product Attention에서 $\frac{1}{\sqrt{d_k}}$을 제거한 것

  → Dot-Product Attention이 시간/메모리 측면에서 효율적

- QK^T : 쿼리와 키 행렬의 내적 계산
- $\frac{1}{\sqrt{d_k}}$: $\sqrt{d_k}$로 나눠 스케일링

  → d_k가 클 경우 내적 값이 커져 sofmax가 매우 작은 값이 되는 것을 방지하기 위함)
- sofmax에 적용한 뒤, 각 값과 가중합 계산

3.2.2. Multi-Head Attention

- 여러 개의 attention funtion을 병렬적으로 수행하는 것
  - multi-head attention을 통해 다양한 부분에 attention 하는 효과
  - 각 head의 결과를 concat해 최종 출력 생성
- 출력:

$$multihead(Q, K, V) = Concat(head_1, ..., head_h)W^O$$

$$where\ head_i = Attention(QW_i^Q, KW_i^K, VW_i^V)$$

- 본 연구에서는 h=8, 즉 8개의 head attention 사용
  - concat을 하면서 차원(dk=dv=dmodel/h)은 64가 됨
  - 결국 각 head의 dimension이 줄어들기 때문에 최종적인 computational cost는 각 single-head attention을 모든 차원에 대해서 할 때와 비슷함

3.2.3. Applications of Attention in our Model

- 본 연구에선 세 종류의 multi-head attention이 사용됨
  - Encoder-Decoder attention
    - Q는 이전 decoder에서, K와 V는 encoder의 output에서 옴
    - 즉 decoder와 encoder의 시퀀스 벡터들이 어떤 correlation을 가지는지 학습
  - Self-attention in encoder
    - Q, K, V 모두 encoder에서 옴
    - 각 postion과 모든 position간의 correlation information을 더해줌
    - 즉, 한 단어(Q)가 모든 단어들(K) 중 어떤 단어들과 correlation이 높고 낮은지 학습
  - Self-attention in decoder
    - 전반적인 프로세스는 encoder와 비슷
    - Auto-regressive property의 보존을 위해 masking vector를 사용하는 것이 차이점

3.3. Position-wise Feed-Forward Networks

- 각 sub-layers는 fully connected feed-forward를 진행
- ReLU 를 활용한 두 가지 linear transformation을 거침

$$FFN(x) = max(0, xW_1 + b_1)W_2 + b_2$$

3.4. Embeddings and Softmax

- 다른 시퀀스 모델과 마찬가지로 input과 output token은 embedding layer를 거쳐서 사용함
- 두 embedding layers와 pre-softmax 선형변환 간에 같은 weight matrix 공유

3.5. Positional Encoding

- Transformer는 input을 순차적으로 입력하지 않으므로 순서 정보가 누락됨

    - 따라서 시퀀스 정보를 데이터에 추가하는 Positional encoding 필요
- sine과 cosie 함수 사용

$$PE(pos, 2i) = sin(pos/10000^{2i/d_{model}})$$

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

    - pos = postion, i = dimension

4. Why Self-Attention

왜 RNN이나 CNN이 아닌 Self-attention을 사용할까?

Table 1: Maximum path lengths, per-layer complexity and minimum number of sequential operations for different layer types. n is the sequence length, d is the representation dimension, k is the kernel size of convolutions and r the size of the neighborhood in restricted self-attention.

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

4.1. Total computational complexity per layer

- Table 1의 Complexity per Layer
    - Self-attention = `O(n^2*d)`, Recurrent = `O(n*d^2)`
    - 보통 n<d 이므로 self-attention의 계산 복잡도가 낮음
- 더 복잡도를 낮추고 싶다면 restricted neighbor(r)만 고려하게 만들 수 있음
    - Self-attention with retriction = `O(r*n*d)`
    - 이때 max path length는 O(n/r)으로 늘어남 → 후속 연구 필요

4.2. Amount of computation that can be parallelized

- Table 1의 Sequential Operations
    - RNN은 input을 순차적으로 입력 → O(n)
    - Self-attention은 input의 모든 position 값들을 연결해 한번에 처리 → O(1)

4.3. Path length between long-range dependencies in the network

- Maximum path length
  - path length = forward와 backward signals 간의 길이
  - 장거리 의존도를 잘 학습하기 위해선 maximum path length가 짧아야 함
- Table 1의 Maximum Path Length
  - Self attention에서 각 토큰(Q)은 모든 토큰(K)을 고려해 correlation 정보 구함
  - 따라서 max path length가 O(1)로 매우 작음 → 장거리 의존도도 더 쉽게 학습

5. Training

5.1. Training Data and Batching

- WMT 2014 영어-독일어 데이터셋: 450만 개의 문장 pair
  - Byte-pair encoding으로 37,000개의 토큰 생성
- WMT 2014 영어-프랑스어 데이터셋 : 3600만 개의 문장 pair
  - 32,000개의 토큰 생성
- each traing batch: 25,000개의 source tokens & 25,000개의 target tokens

5.2. Hardware and Schedule

- 8 NVIDIA P100 GPU로 트레이닝
- Base model: step time은 0.4초, 총 100,000steps으로 12시간 소요
- Big model: step time은 1.0초, 총 300,000 steps로 3.5일 소요

5.3. Optimizer

- Adam optimizer 사용

$$\beta_1 = 0.9,\ \beta_2 = 0.98,\ \epsilon = 10^{-9}$$

- 아래 공식에 따라 learning rate 변화

$$lrate = d_{model}^{-0.5} \times min(step\ num^{-0.5},\ step\ num \times warmup\ steps^{-0.5})$$

  - 첫번째 warmup steps에서는 lrate를 선형적으로 증가시킴

- 이후에는 step num의 역제곱근에 비례하게 lrate 감소시킴
- warm up steps = 4,000

5.4. Regularization

Residual Dropout

- sublayer의 출력에 드롭아웃 적용했다.
- 인코더와 디코더 stack에서 임베딩과 위치 인코딩의 sum에 드롭아웃 적용했다.

Lable Smooting

- 라벨 스무딩 적용 (훈련 데이터의 정답 라벨을 약간 수정해서 모델이 정답을 100프로 확신하지 않도록 하는 기법.) > perplexity를 악화시키지만 모델이 더 일반화된 방식으로 학습하게 되어서 정확도와 BLEU 점수를 향상시켰다.

6. Result

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**	$2.3 \cdot 10^{19}$	

6.1. Machine Translation

영어-독일어 / 영어-프랑스어 번역에서 Transformer이 다른 모델들에 비해 높은 성능을 가지면서 training cost도 낮다.

6.2. Model Variations

	N	d_{model}	d_{ff}	h	d_k	d_v	P_{drop}	ϵ_{ls}	train steps	PPL (dev)	BLEU (dev)	params $\times 10^6$
base	6	512	2048	8	64	64	0.1	0.1	100K	4.92	25.8	65
(A)				1	512	512				5.29	24.9	
				4	128	128				5.00	25.5	
				16	32	32				4.91	25.8	
				32	16	16				5.01	25.4	
(B)					16					5.16	25.1	58
					32					5.01	25.4	60
(C)	2									6.11	23.7	36
	4									5.19	25.3	50
	8									4.88	25.5	80
		256			32	32				5.75	24.5	28
		1024			128	128				4.66	26.0	168
			1024							5.12	25.4	53
			4096							4.75	26.2	90
(D)							0.0			5.77	24.6	
							0.2			4.95	25.5	
								0.0		4.67	25.3	
								0.2		5.47	25.7	
(E)	positional embedding instead of sinusoids									4.92	25.7	
big	6	1024	4096	16			0.3		300K	**4.33**	**26.4**	213

영어 독일어 번역

attention heads와 attention keys 와 value dimensions 등을 바꿔가면서 성능에 어떤 영향을 주는 지 보는 실험이다.

(결과)
(B)열에서
d_k를 너무 줄이면 모델 성능이 안좋아진다.

(C)열에서 모델 사이즈가 클수록 성능이 더 좋다.

(D)열에서 drop out 이 오버피팅을 피하는 데 도움이 된다.

Conclusion

sequence transduction task에 대해 인코더-디코더 구조에서 그동안 자주 사용된 recurrent layers 대신 multi-headed self attention을 사용한 모델 Transformer을 소개했다. translation tasks에서는 CNN이나 RNN보다 transformer이 더 빠르게 학습될 것이다.