Attention is All You Need 논문 리뷰 - team2

윤채영

Abstract

- 기존의 sequence transduction(기계 번역, 답변 생성, 요약 등) model

  : RNN or CNN 기반 + encoder와 decoder 연결에 Attention mechanism 사용

- **Transformer** architecture 제안

  : **Attention mechanism만** 사용 (w/o RNN or CNN)

1. Introduction

- **RNN**의 특징
  - input과 output seq의 각 **단어 위치**에 따라, **factor** computation 수행 → 문장을 나눠서, 순서대로 처리
  - computation step에 따라, hidden state h_t의 sequence를 순서대로 생성
  - $h_t = f(h_{t-1}, x_t)$ → **t 시점**의 은닉 상태 계산을 위해서는, **t-1 시점**의 은닉 상태가 필요 → 본질적으로 **병렬 처리가 불가능한 구조** ! → 연산 속도 **느림**
  - 성능 개선을 위한 노력: factorization tricks & conditional computation
- **Transformer**
  - Attention mechanism 적용: input과 ouput seq의 **거리와 상관없이**, **단어들 사이의 관계를 모델링**할 수 있음 (↔ **RNN**: 거리가 먼 단어들끼리의 관계를 잘 잡지 못함)
  - recurrenct 구조 미사용 & **오직** Attention mechanism**만** 사용

✅ **global dependency**(멀리 떨어진 단어들 사이의 관계)까지도 잘 포착

✅ **병렬 처리**(parallelization) 가능

2. Background

[RNN의 한계 발견] - 순차 처리 느림 & 장거리 의존성 학습 어려움

↓ (한계 극복 시도)

[대안 1. CNN 기반 모델 등장] - 병렬 처리 가능 & 속도 향상 but 장거리 단어 간 연산량 증가

&

[대안 2. End-to-End Memory Networks] - recurrent attention 구조

↓ (중요한 전환점)

[💡 Transformer 등장] - CNN, RNN 없이 self-attention만 사용

- **CNN-based model**들의 등장
  - RNN의 가장 큰 문제점인, sequential computation으로 인한 **[병렬 처리 불가 & 속도 저하 문제]를 해결**하기 위함
  - 모델 예시: Extended Neural GPU, ByteNet, ConvS2S 등
  - 장점) 각각의 모든 position에 대해, hidden representation을 **병렬적으로 연산 가능**
  - 단점) **멀리 떨어진 단어들 간의 관계 학습이 어려움** - position 간의 distant에 비례해서 연산량이 증가하기 때문
    - ConvS2S: 연산량이 두 위치 간 거리에 대해 linearly 증가
    - ByteNet: 연산량이 두 위치 간 거리에 대해 logarithmically 증가
- **End-to-end memory networks**
  - RNN의 대안으로서 등장 - sequence-aligned RNN 미사용
  - **recurrent attention mechanism** 기반: attention을 반복적으로(recurrently) 여러 번 실행 → 문장 내 중요한 정보에 여러 차례 집중
  - 활용 분야: 간단한 질문 답변 / 언어 모델링
- **Transformer**
  - 두 단어 간 거리와 상관없이, 관계 학습을 위한 **연산량이 일정**함 !
  - 하지만, Attention의 처리 방식(각 위치에 attention score로 가중치를 주어서 평균을 냄)으로 인한 **정보의 해상도 저하** 문제

    → Multi-Head Attention 기법으로 보완
- **Self-attention** ⭐
  - **하나**의 sequence(문장) 내, **서로 다른 위치**의 단어들 간 관계를 계산하는 메커니즘
  - sequence(문장) 전체를 더 잘 이해할 수 있는 표현(vector)을 만들기 위함
  - 활용 분야
    - reading comprehension(독해)
    - abstractive summarization(추상적 요약)
    - textual entailment(문장 간 관계 파악)
    - task-independent sentence representations(문장 표현 학습)

> 💡 **Transformer**: RNN, CNN 등의 순서 기반 신경망 없이, **완전히 self-attention에만 의존**한 모델 !

3. Model Architecture

- **기본적인 Encoder-Decoder 구조**
  - **Encoder**: input 시퀀스 $\mathbf{x}$를 받아서 → continuous representation(숫자 벡터)의 시퀀스 $\mathbf{z}$로 변환
  - **Decoder**: 시퀀스 $\mathbf{z}$를 받아서 → output 하나씩 생성 (y_1 → y_2 → y_3 ...) → output 시퀀스 $\mathbf{y}$ 생성
  - **Auto-Regressive** 방식: 이전 결과 y_t를, 다음 단어 y_{t+1} 생성을 위한 추가적인 input으로 사용
- **[Transformer의 전체 architecture]**

  : Encoder와 Decoder 각각에 stacked self-attention과 point-wise fc layers 사용

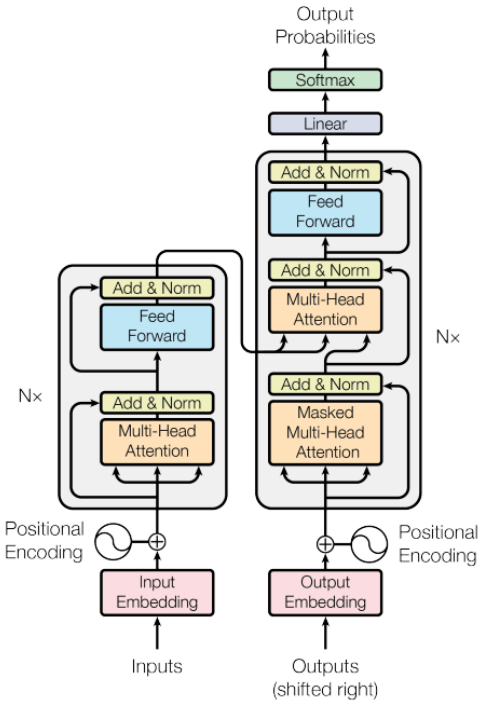


Figure 1: The Transformer - model architecture.

3.1 Encoder and Decoder Stacks

- **Encoder**
  - 총 6개의 identical한 layer들로 구성
  - 각 layer는 2개의 SubLayer로 구성
    1. **Multi-Head Self-Attention mechanism**
    2. **Position-wise Feed-Forward network**
       - 각 SubLayer에 `residual connection` + `layer normalization` 적용
       - 각 SubLayer의 output: $\mathrm{LayerNorm}(x + \mathrm{Sublayer}(x))$
  - model의 모든 layer의 **출력 차원 = 512**로 통일 → 잔차 연결을 위함
- **Decoder**
  - 총 6개의 identical한 layer들로 구성
  - 각 layer는 3개의 SubLayer로 구성
    1. Multi-Head Self-Attention mechanism → **Masking 적용 !**
    2. **Multi-Head Attention over the output of the Encoder stack**
    3. Position-wise Feed-Forward network
       - 각 SubLayer에 `residual connection` + `layer normalization` 적용
  - Decoder에서 i 시점의 단어를 예측할 때, i - 1 시점까지의 결과만 보도록 하는 방법
    1. SubLayer 1에 **Masking**을 적용 → 이후 시점의 단어들을 참고하지 못하도록 masking
    2. output embedding의 위치를 **한 칸 밀기(offset)** → i - 1 시점까지의 단어만 Decoder에 입력됨

강서연

3.2 Attention

Attention 함수: 간단하게 query와 key-value 쌍을 출력으로 내보는 것.

input: **query, key, value**

output: query와 key 간의 유사도를 계산하는 **compatibility function**을 이용한 가중치를 이용해 가중합으로 나타남

3.2.1 Scaled Dot-Product Attention

이 논문에서 사용하는 attention → Scaled Dot-Product Attention

- input:
  - query: d_k 차원으로 모든 query를 하나의 행렬 Q로 표현 → 유사도 계산에 사용
  - keys: d_k 차원으로 모든 key를 하나의 행렬 K로 표현 → 유사도 계산에 사용
  - values: d_v차원으로 모든 value를 하나의 행렬 Q로 표현 → 실제 정보 생성에 사용
- ⇒ $Attention(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$
- 자주 사용되는 Attention
  - **Additive attention**: **compatibilty function**(query, key간의 유사도 평가)을 계산하며 d_k로 scaling을 하지 않았을 때 더 나은 성능을 보임
  - **Dot-Product attention**: 위에서 언급한 attention 식과 동일하지만 scaling factor $\frac{1}{\sqrt{d_k}}$ 적용
    - 장점) 내적 연산을 하기 때문에 더 빠르고 효율적으로 처리 가능
    - 단점) d_k차원 증가(고차원 연산일때)
      → q,k는 평균이 0이고 분산이 1인 random variable이라고 했을 때 둘의 내적값 - 평균 0 분산 d_k
      → 내적 값 간 큰 차이 (분산이 큼)
      → softmax 함수 적용 시 하나의 값으로 치우침
      →
      매우 작은 기울기 가지기에 가중치 업데이트에 문제
    - ⭐ **문제 해결을 위해 $\frac{1}{\sqrt{d_k}}$로 scaling**

3.2.2 Multihead Attention

하나의 단일한 single attention 을 사용하는 것보다 **여러 개의 attention**을 사용하는 Multihead Attenntion 이 더 효과가 좋음

- 방법: 각 버전의 query, key, value들을 병렬적으로 두고 계산한 뒤 concatenate 하여 다시 한번 project되어 사용
  → 이를 통해 서로 다른 representation subspace에 대해 학습하여 일반적인 학습이 가능
  ** 서로 다른 표현 학습 가능, rich한 표현 생성
- 식: $MultiHead(Q, K, V) = \text{Concat}(head_1, \ldots, head_h)W^O$
- 실제 사용: h=8인 병렬 layer을 사용했고 (8개의 multihead 존재), 각 version 마다 $d_k = d_v = \frac{d_{model}}{h} = 64$의 차원을 동일하게 가짐
  - h의 개수 만큼 각 version의 attention의 차원을 줄임으로써 전체적인 연산비용은 비슷하게 가져감

3.2.3 Applications of Attention in our Model

multi-head attention 의 적용 방법

1. encoder-decoder attention 에서 query는 이전의 decoder layer에서, key와 value는 encoder의 출력을 가져옴 → decoder가 입력 시퀀스의 모든 위치를 이용(attend, 주의를 기울임) (= seq-to-seq와 유사)

2. encoder가 `self-attention layer` 을 가짐
   → encoder가 이전 layer의 encoder의 모든 위치를 이용(attend)
3. decoder 역시 `self-attention layer` 을 가짐 [[⭐미래의 정보를 decode 하는데 사용하면 안됨]]
   → masking을 통해 자기회귀 특징을 지키면서, 이전 정보의 모든 위치를 이용(attend)

백서현

3.3 **Position-wise Feed-Forward Networks**

> **단어별로 독립적으로 각** 단어의 정보를 정제하는 신경망. self-attention에서 문맥 정보를 반영했다면, FFN에서는 각 단어 벡터를 정교하게 처리한다.

- 구조 : 입력 벡터 → Linear (W_1, b_1) → ReLU → Linear (W_2, b_2) → 출력 벡터
  - 즉, $FFN(x) = max(0, xW1 + b_1)W_2 + b_2$
    - ReLU = $max(0, x)$
  - 먼저 선형 변환을 통해 x를 더 넓은 차원으로 확장했다가 ReLU로 비선형성을 추가하고, 다시 선형변환하여 차원을 축소한다. (ex) 512 → 2048 → 512차원) 이때, 입력과 출력의 차원(d_{model})은 512로 같으며, 중간에 가중치 행렬에 의해 d_{ff} =2048 차원으로 확장되는 것이다.
- encoder와 decoder 모두에 존재한다. encoder, decoder 각각에 있는 N개의 layer는 각각 다른 parameter를 사용하며, 같은 layer에서는 모든 단어 위치에 대해 같은 parameter를 사용한다.
- FFN은 종속성 없이 단어마다 독립적으로 작동하므로 “two convolutions with kernel size 1”라고도 불린다. 즉 단어벡터 하나씩만 보고 처리하는 과정을 두 번(선형변환)한다는 것이다.

3.4 Embeddings and Softmax

- 다른 시퀀스변환 모델 (..) 과 마찬가지로 임베딩 사용 : 입출력 토큰을 d_{model} 크기의 벡터로 변환 ex) (I) → [0.2, 0.4, ..., 0.7] 512차원
- decoder 마지막 단계에서 선형변환 → softmax 함수로 출력벡터를 예측되는 다음 토큰의 확률로 변환
  - linear 변환 : softmax에 넣을 logit을 만드는 변환. decoder에서 생성된 벡터를 훨씬 더 큰 벡터로 투영한다. ex) 모델이 학습 데이터셋에서 학습한 10000개의 고유한 영어 단어를 알고 있으면 10000개 셀로 구성된 Logit vector을 만들며, 각 셀은 단어의 score를 담고 있다.
  - softmax : 모든 단어들에 대한 score를 확률로 변환한다. 가장 높은 확률값이 다음단어!
- 두 embedding layer (encoder에 처음 단어를 정수에서 고차원 벡터화시키는 Layer와, 출력 전에 decoder의 출력 벡터를 단어로 변환하는 선형 변환 layer를 뜻함) 는 같은 가중치행렬을 쓴다.
  - embedding layer들에서는 이 가중치행렬에 $\sqrt{d_{model}}$ 을 곱한다.
  - embedding이 너무 작은 값이 되면 학습이 안될 수 있기 때문에 스케일링하는 작업!

3.5 Positional Encoding

> 🧨 RNN, LSTM과 달리 입력되는 문장을 순차적으로 처리 하지 않고, 입력된 문장을 병렬로 한번에 처리함. 연산은 빠르지만 단어의 위치(순서)를 알수 없는 문제 → 위치를 나타내는 인코딩을 추가하자!

- positional encoding의 차원 : 입력차원과 같음 (d_{model})
- 모든 위치값은 시퀀스 길이, input에 상관없이 동일한 식별자를 가져야한다. 즉 시퀀스 변경되어도 positional encoding은 동일하다.
- 논문에서 제시한 positional encoding

  이렇듯 첫번째 차원부터 마지막 차원의 벡터값을 서로 다른 frequency를 가진 sin & cos을 번갈아 계산하면서 채우면 충분히 다른 encoding 값을 지닐 수 있게 된다.

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

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

  - pos : position (몇번째 단어인지) / i: 차원 (단어 임베딩에서 몇번째 차원인지)
- 이때 주기함수의 파장(wavelength)은 2π부터 10000·2π까지 기하급수적 증가. 즉 뒷쪽 차원으로 갈수록 (i가 커질수록) 파장이 길어진다. 이는 곧 뒷쪽 차원으로 갈수록 위치가 많이 달라져도 출력값이 비슷하게 유지되어, 전체 구조에서 거시적인 위치 패턴을 인식할 수 있다는 뜻이다.
- ⚠️ 꼭 sin,cos 기반 고정 인코딩이 아니더라도 위치정보를 학습하는 positional embedding도 잘 작동한다.
- 그러나 논문에서 sin,cos 기반 인코딩 선택한 이유는 두가지다. 첫째, 위치 차이 k로 고정되어있을때 pos+k 위치의 positional encoding을 pos 위치의 positional encoding으로 계산 가능하다. 즉 상대적 위치 정보 학습이 쉽다. (ex) 단어 A와 B 사이 3칸 차이가 있다고 하면, A positional encoding을 알면 B도 선형적으로 계산 가능) 둘째로, 학습 중 보지 못한, 길이가 더 긴 문장에서도 일반화될 수 있기 때문이다.

4. Why Self-Attention

왜 RNN, CNN 계열이 아닌 Self-Attention을 사용해야 하는가?

1. 각 레이어마다 계산복잡도가 줄어든다. 즉, 연산을 많이 하지 않는다.
2. 단어 순서 따라 계산해야하는 recurrence를 없애서 병렬적 처리가 가능하다.
3. long-range dependency를 잘 처리할 수 있다. 시퀀스 변환 task에서는 신호가 앞뒤로 이동할때 통과하는 경로가 짧아야, 앞에 있는 단어와 뒤에 있는 단어 사이의 관계(장거리 의존성) 를 모델이 잘 기억할 수 있다.

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)	$O(r \cdot n \cdot d)$	$O(1)$	$O(n/r)$

n: 시퀀스 길이(단어수) / d : d_{model}(임베딩 차원) / k : CNN 커널 사이즈 / r : self-attention이 참조하는 주변 토큰 수

- 계산복잡도 : n<d일 때 Self-Attention이 RNN보다 빠르다. 대부분의 경우 시퀀스길이보다 임베딩 차원이 크므로, self-attention이 더 빠르다는 뜻.
- 순차연산 : Self-Attention 은 한번만에 연산할 수 있지만 (병렬처리), RNN은 시퀀스 길이만큼 연산이 필요하다.
- Self-Attention (restricted)는 시퀀스가 너무길때 전체에 attention을 적용하지 않고 출력위치 주변의 r개 단어만 보는 layer이다. 이럴 경우 self-attention의 최대 경로 길이가 $O(1)$에서 $O(n/r)$로 늘어난다.

  ex) 1번 단어가 2~6번까지밖에 못 보는데 10번 단어와 정보 교환하려면 → 여러 단계 필요
- CNN에서 k<n이면 하나의 CNN layer로 모든 입-출력 위치를 연결 불가능

ex) 1번 단어는 1~3번 단어까지만 보게되면 , 1번 단어의 정보가 8번 단어까지 바로 닿지 않아 전범위 정보 전파 X.

→ 모든 층을 연결하려면 여러 층이 필요함! 일반적으로는 $O(n/k)$ 이나 maxiumum path length를 늘리려면 확장된 CNN은 $O(\log_k n)$ 층이 필요하다.

- 또한 계산복잡도 측면에서, CNN이 일반적으로 RNN보다 계산비용이 k만큼 크다 (필터를 여러 위치에 적용하므로)
- Seperable Convolution은 계산복잡도를 기존 CNN의 $O(knd^2)$에서 $O(knd + nd^2)$ 로 크게 줄일 수 있다. 그러나 k = n이 되더라도 결국 복잡도는 Self-Attention + FeedForward Layer와 비슷해져서, 굳이 CNN 쓸 이유 X
  - Seperable Convolution : **채널 별로 필터 적용**하고, 나중에 결과를 합치는 방식.

이외 특징

- 어떤 단어를 주의하고 있는지 해석 가능성이 더 높다. (ex) heatmap 시각화)
- 개별 attention head들은 서로 다른 역할을 함

  ex) head 1은 동사-주어 관계를 집중해서 보고 head2는 명사구 전체에 집중

김종현

5. Training

5.1) Training Data and Batching

- **학습 데이터**
  - **영어-독일어 (En-De)**: WMT 2014, 약 **450만 문장 쌍**
  - **영어-프랑스어 (En-Fr)**: WMT 2014, 약 **3600만 문장 쌍**
- **토크나이징 방법**
  - En-De: **BPE (Byte Pair Encoding)** 사용 (단순 빈도순으로 tokenizing)

    → **출력-입력 단어 공유 vocabulary**, 약 **37,000개 토큰**
  - En-Fr: **WordPiece 토크나이저**로 분할 (빈도 뿐만 아니라 liklihood 이용해 의미추출), **32,000개 vocabulary**
- **배치 구성**
  - 문장 길이를 기준으로 비슷한 문장들을 배치로 묶음
  - 각 배치에는 약 **25,000개의 입력 토큰** + **25,000개의 출력 토큰** 포함 (토큰 수는 하이퍼파라미터)

5.2) Hardware and Schedule

- **훈련 환경**:
  - GPU: **1대의 머신, 8개의 NVIDIA P100 GPU 사용**
- **Base 모델**:
  - 학습 속도: **0.4초 / 스텝**
  - 총 학습 횟수: **100,000 스텝**
  - 총 학습 시간: **약 12시간**

- **Big 모델**:
  - 학습 속도: **1.0초 / 스텝**
  - 총 학습 횟수: **300,000 스텝**
  - 총 학습 시간: **약 3.5일**

✅ 하이퍼파라미터 정리

하이퍼파라미터	설정 값	설명
Number of layers (N)	6	인코더 6층 + 디코더 6층
Hidden size (d_model)	512	각 벡터의 차원 수
Feedforward size (d_ff)	2048	FFN 내부의 중간 차원
Number of heads	8	멀티 헤드 어텐션의 head 개수
Dropout	0.1	attention, FFN, embedding에 적용
Positional Encoding	Fixed (sinusoidal)	학습하지 않음
Attention dropout	0.1	어텐션 가중치에 dropout 적용
Label smoothing	0.1	학습 시 정답 분포를 부드럽게
Optimizer	Adam	β1=0.9,β2=0.98,ϵ=10e−9
Learning rate schedule	Warm-up + decay	warm-up steps = 4000, 이후 inverse square root decay
Batch size	약 25,000 tokens (입력+출력 합산)	토큰 수 기준 배치
Vocabulary size	37,000 (En-De), 32,000 (En-Fr)	BPE/WordPiece 기반

5.3) Optimizer

🔷 Learning Rate

$$lrate = d_{\text{model}}^{-0.5} \cdot \min(step\_num^{-0.5}, step\_num \cdot warmup\_steps^{-1.5})$$

> 처음에 우항을 기준으로 min 값 설정됨 (점차 증가) 이후 step_num 커져감에 따라 좌변으로 조정되며 감소

🔷 의미 요약

구간	동작 방식
처음 warmup_steps (= 4000)	학습률을 **선형 증가** (step 수에 비례)
이후	학습률을 **역제곱근 감소** (step$^{-0.5}$ 비율로 감소)

→ 이 방식은 초반에는 빠르게 학습을 안정화시키고,

이후에는 학습률을 점점 줄이면서 **안정된 수렴**을 유도.

Adam Optimizer 공식 (Kingma & Ba, 2014)

Adam은 **모멘텀(Momentum)**과 **RMSProp**의 장점을 결합한 최적화 알고리즘.

---

1. **초기화**

$m_0 = 0$, $v_0 = 0$ (moment vector, squared gradient vector)

2. **t번째 스텝에서 기울기 계산**

$g_t = \nabla_\theta L_t$ (현재 파라미터 θ에 대한 손실 함수의 그래디언트)

3. **모멘텀(1차 추정치) 업데이트**

$$m_t = \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$$

4. **2차 모멘텀(기울기 제곱) 업데이트**
   변동성이 너무 크면 좋지 않음

$$v_t = \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$$

5. **편향 보정 (Bias correction)**
   후반으로 갈 수록 학습이 더 의미가 있어짐

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}, \quad \hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

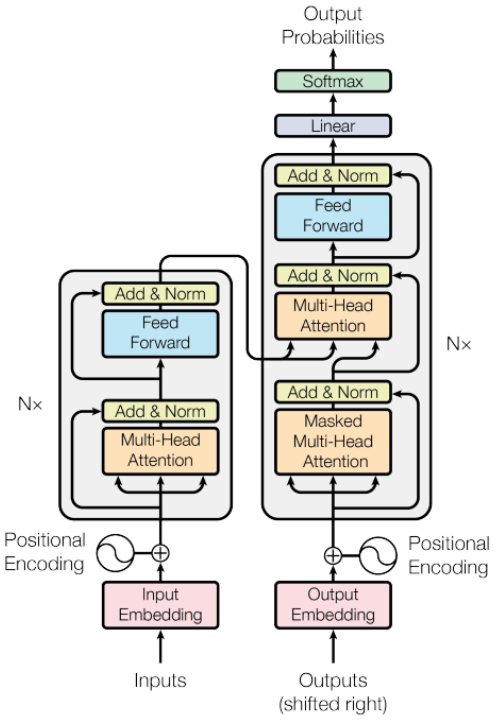
6. **파라미터 업데이트**

$$\theta_t = \theta_{t-1} - \eta \cdot \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

5.4) Regularization

🔷 1. **Residual Dropout**

- 적용 레이어
  - 각 **서브 레이어 출력** (예: Self-Attention, FFN) → **add & norm 전에 dropout**
  - **입력 임베딩 + positional encoding의 합**에도 dropout 적용 (인코더/디코더 둘 다)
- **Dropout 비율:**Pdrop=0.1



🔷 2. **Label Smoothing**

- **정답 레이블을 100% 확신하지 않도록 부드럽게 처리**
  - 예: 원래 one-hot label → 정답: 0.9, 나머지: 0.1 분산
- **값:**εls=0.1
- **효과:**
  - **Perplexity는 약간 나빠짐** (불확실하게 학습되므로)

$$\text{PPL} = e^{\text{CrossEntropy}}$$

  - 하지만 **정확도와 BLEU 점수는 향상**됨 (일반화 성능 향상)

$$\text{BLEU} = \text{Brevity Penalty} \cdot \exp\left(\sum_{n=1}^{N} w_n \log p_n\right)$$

- p_n: n-gram precision
- w_n: 가중치 (보통 uniform)

- **n-gram precision** (단어 일치 + 어순 고려)
- **Brevity Penalty** (너무 짧은 문장은 감점)

6 .Results

6.1) Machine Translation

🔷 English-to-German 번역 성능

- **Transformer (big)** 모델:
  - BLEU 점수 **28.4**
  - 기존 최고 성능(앙상블 포함)보다 **2.0 BLEU 이상 향상**
  - 학습 시간: **3.5일 (8×P100 GPU 사용)**
- **Transformer (base)** 모델도:
  - 이전 모든 모델과 앙상블보다 성능 우수
  - **훨씬 적은 학습 비용**으로 달성

🔷 English-to-French 번역 성능

- **Transformer (big)** 모델:
  - BLEU 점수 **41.0**
  - 기존 최고 단일 모델보다 더 우수
  - 학습 비용: **이전 SOTA 대비 1/4 수준**
  - 특이사항: dropout rate은 **0.1 (기존 0.3보다 낮춤)**

학습 및 추론 설정

항목	설정값
Checkpoints averaging	Base: 마지막 5개 평균Big: 마지막 20개 평균
Beam search	beam size = 4, length penalty α=0.6
출력 최대 길이	input length + 50 (단, 조기 종료 허용)
학습 FLOPs 추정 방식	training time × GPU 수 × GPU FLOPs 처리량

6.2) Model Variations

전체 실험 개요

- 목적: **Transformer 구성 요소**의 중요성을 평가
- 실험 기준:
  - 데이터셋: **WMT 2014 English-to-German**
  - 개발셋: **newstest2013**
  - 평가지표: **Perplexity (PPL)**, **BLEU (dev 기준)**, **파라미터 수**
  - **Beam search 사용**, **Checkpoint averaging은 사용하지 않음**

Table 3: Variations on the Transformer architecture. Unlisted values are identical to those of the base model. All metrics are on the English-to-German translation development set, newstest2013. Listed perplexities are per-wordpiece, according to our byte-pair encoding, and should not be compared to per-word perplexities.

	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

🔷 (A) Head 수, Key/Value 차원 변화 실험

- head 수: 1, 4, 8, 16, 32
- **너무 적거나 많아도 성능 저하**
  - 1 head → BLEU 24.9 (↓0.9)
  - 8 heads → BLEU 최고 25.8
  - 32 heads → 성능 하락 (BLEU 25.4)

→ 적정한 head 수(8~16)가 중요, 너무 많아도 과분산으로 성능 저하

🔷 (B) Key dimension (d_k) 축소 실험

- d_k 줄이면 **성능 하락** (BLEU 25.1~25.4)
- 의미: **어텐션 상의 유사도 계산이 단순하지 않음**

  → dot product 외의 복잡한 방식이 유리할 수도 있음

🔷 (C) 모델 크기 확장 실험 (d_model, d_ff 증가)

모델 크기	BLEU
d_model=1024 → BLEU ↑ 26.0	
d_ff=4096 → BLEU ↑ 26.2	

→ 모델을 키우면 성능 향상, 특히 d_ff 크기 증가가 유효

🔷 (D) Dropout 비율 변화

- Dropout이 **0.0**이면 **성능 크게 하락**

- PPL ↑, BLEU ↓
- 0.1~0.2가 적절

  → **Overfitting 방지에 필수**

🔷 (E) Positional Encoding 방식 변경

- 기존: **sinusoidal (고정값)**
- 실험: **learned positional embedding**
- 결과: **거의 동일한 성능** (BLEU 25.7 ≈ 25.8)

→ 학습된 positional encoding도 충분히 사용 가능함 (실제로 BERT에서는 사용)

6.3) English Constituency Parsing

✅ 1. 실험 목적 및 과제 특성

> Transformer의 범용성(generalization)을 확인하기 위해
>
> **영어 구문 분석(Constituency Parsing)** 실험을 수행.

- 이 과제는 일반 번역보다 어렵다:
  - **출력이 입력보다 훨씬 길다**
  - **문법적 제약(구조)이 강하다**
  - RNN 기반 seq2seq 모델은 **데이터가 적을 때 성능이 낮았다** ([37] 참고)

✅ 2. 모델 구성 및 데이터

설정 항목	내용
모델 구조	**4-layer Transformer**, dmodel=1024
학습 데이터	• **WSJ only (Wall Street Journal, Penn Treebank)**: 약 4만 문장 • **Semi-supervised setting**: Berkeley Parser 등에서 얻은 **17M 문장** 추가 | Vocab size | WSJ only: 16K / semi-supervised: 32K |

✅ 3. 하이퍼파라미터 및 추론 설정

항목	설정
Dropout / learning rate / beam size	Section 22 dev set에서 소수 실험으로 설정
나머지 하이퍼파라미터	English-German translation base 모델과 동일
Beam size	21
Length penalty α\alphaα	0.3
Max output length	input length + 300

Table 4: The Transformer generalizes well to English constituency parsing (Results are on Section 23 of WSJ)

Parser	Training	WSJ 23 F1
Vinyals & Kaiser el al. (2014) [37]	WSJ only, discriminative	88.3
Petrov et al. (2006) [29]	WSJ only, discriminative	90.4
Zhu et al. (2013) [40]	WSJ only, discriminative	90.4
Dyer et al. (2016) [8]	WSJ only, discriminative	91.7
Transformer (4 layers)	WSJ only, discriminative	91.3
Zhu et al. (2013) [40]	semi-supervised	91.3
Huang & Harper (2009) [14]	semi-supervised	91.3
McClosky et al. (2006) [26]	semi-supervised	92.1
Vinyals & Kaiser el al. (2014) [37]	semi-supervised	92.1
Transformer (4 layers)	semi-supervised	92.7
Luong et al. (2015) [23]	multi-task	93.0
Dyer et al. (2016) [8]	generative	93.3

$$\mathrm{F1} = \frac{2 \cdot \mathrm{Precision} \cdot \mathrm{Recall}}{\mathrm{Precision} + \mathrm{Recall}}$$

- **Precision**: 모델이 예측한 constituent 중 정답과 일치한 비율
- **Recall**: 정답 constituent 중 모델이 맞춘 비율

- Transformer는 **복잡하고 구조적인 과제인 구문 분석**에도 잘 일반화됨
- **semi-supervised 설정에서 F1 = 92.7**, 대부분의 이전 모델보다 우수
- **task-specific 구조 변경 없이**도 성능이 좋았다는 점에서 의미 있음
- **RNN 기반 seq2seq보다 우수**, **Berkeley Parser보다도 우수**