Attention is all you need

Abstract

가장 지배적인 변환(transduction) 모델은 인코더와 디코더를 포함하고 있는 RNN 또는 CNN에 기반하고 있다. 가장 우수한 모델은 어텐션 메커니즘을 통해 인코더와 디코더를 연결한 모델이다. 저자와 연구팀은 순환성이나 합성곱(Convolution) 연산을 제거하고 오직 어텐션 메커니즘에 기반한 새로운 네트워크 구조를 제시하고자 한다. 우리는 이 모델을 Transformer라고 부른다. 두 개의 기계 번역 태스크에서 이 모델은 병렬 계산이 가능하다는 것을 보여주었고 크게 훈련 시간을 단축시켰을 뿐만 아니라 성능에서도 우수함을 보여주었다.

Instruction

LSTM이나 GRU를 포함한 RNN 모델은 sequence modeling과 Language modeling과 기계 번역과 같은 변환 문제에서 최적의 접근 방법으로 자리매김하였다. 그 후 순환 모델이나 인코더-디코더 구조를 넘어서려는 수 많은 노력들이 있었다.

순환 모델(Recurrent Model)은 대개 연속적인 입력 데이터와 출력 데이터의 위치를 따라 연산을 수행한다. 일렬로 연산 순서를 나열하면서 모델은 은닉층의 값들을 출력한다. 즉 이전 시기의 은닉층의 값과 현재의 입력 값을 통해 현재의 은닉층의 값을 출력한다. 이런 연속성을 갖는 모델의 특성 때문에 훈련 시 병렬 연산이 어려워진다. 연산 순서가 긴 데이터일수록 병렬 연산이 더욱 어려워진다. 이후 이 문제를 개선하고자 하는 기법들이 나왔지만 근본적으로 연속적인 연산 수행의 제약에서 완전히 벗어나진 못하였다.

어텐션(Attention) 메커니즘은 여러 task에서 강력한 sequence modeling과 변환(transduction) 모델의 필수적인 요소가 되었다. 어텐션 메커니즘은 연속적인 입력 데이터와 출력 데이터 간의 거리와 무관하게 데이터 간의 연관성을 찾을 수 있게 해준다. 그러나 어텐션 메커니즘은 순환 모델에서 인코더와 디코더의 접합부로 많이 사용된다.

이 논문에서는 Transformer라는 모델을 제시한다. 이 모델의 구조는 순환성을 없애고 전적으로 입력 데이터와 출력 데이터와의 전역적인 연관성을 이끌어내는 어텐션 메커니즘에 의존한다. 이 모델은 병렬 연산 수행을 가능하게 하고 비교적 적은 시간의 훈련으로 높은 성능을 냈다.

Background

ByteNet, ConvS2S 등과 같은 모델은 입력 데이터와 출력 데이터에서 연관성을 찾고자 하는 데이터 간의 거리가 멀수록 연산의 수가 증가한다. 이는 거리가 먼 데이터 간의 연관성을 학습하기 어렵게 한다. ConvS2S는 선형적으로, ByteNet은 지수적으로 증가한다. 하지만 Transformer는 상수 시간으로 고정된다.

Self-attention은 하나의 연속성 데이터에서 데이터 내부에서 다른 위치에 있는 데이터 간의 연관성을 찾는 어텐션 메커니즘이다. 여러 task에 성공적으로 사용되고 있다.

Model architecture

가장 경쟁력 있는 연속성 데이터를 처리하는 신경망 모델은 인코더-디코더 구조를 갖는다. 인코더는 입력 데이터에서 연속적인 데이터를 함축한 값을 출력하고 디코더는 그 값을 입력 받아 한 번의 타임 스텝마다 하나의 값을 출력함으로 연속적인 데이터를 순서에 맞게 출력한다. 각 step에서 모델은 다음 step의 데이터를 생성할 때 이전에 생성된 데이터를 추가 입력 데이터로 사용한다. 즉 auto-regressive하다.

Transformer는 이러한 전체적인 구조를 따르는데 인코더와 디코더에 self-attention과 point-wise, fully connected layer를 겹겹이 쌓아 구성한 구조로 이루어져 있다.

Encoder

인코더는 6개의 동일한 층을 쌓아 올려진 형태로 구성되어 있다. 각 층은 두 개의 부(sub) 층으로 구성되어 있다. 첫번째는 multi-head attention 메커니즘이고 두번째는 position-wise, fully connected feed-forward 네트워크이다. 또한 residual connection을 적용하고 그 결과를 Layer Normalization 한다. 각 부(sub) 층의 결과는 LayerNorm(x + SubLayer(x))가 된다. Residual connection을 구현하기 위해 입력 데이터와 출력 데이터의 shape을 같게 하였다.

Decoder

디코더 또한 6개의 동일한 층으로 쌓아 올려진 형태로 구성되어 있다. 각 인코더의 층에서 두개의 부(sub) 층과 더불어 디코더에는 세번째 층이 추가 되었는데 이 층에서는 인코더의 출력값에 대해 multi-head attention을 수행한다. 인코더와 비슷하게 residual connection을 수행하고 Layer Normalization을 수행한다. 디코더 층에서 첫번째 부(sub) 층에서는 self-attention mechanism을 수정하였는데 디코더가 출력하는 데이터가 각 위치에서 이후 위치에 있는 데이터에 집중(attention)하지 못하게 하도록 하였다. 즉 RNN과 같이 이전 스텝의 데이터를 가지고 다음 스텝의 데이터를 출력하도록 하기 위함이고 출력된 데이터 또한 연속성을 갖게 된다. 이를 위해 making을 사용한다. 이는 위치 i에 데이터를 예측하기 위해 이미 알고 있는 데이터 즉 i보다 이전 데이터만 사용한다.

Attention

어텐션 함수는 쿼리(query)와 key : value 쌍의 집합이 출력값(output)에 맵핑되는 걸로 설명할 수 있다. 여기서 query, key, value, output은 모두 벡터이다. Output은 query, key를 통해 구한 가중치를 value에 곱한 값의 합이다.

Scaled Dot-Production Attention

Transformer에서는 Scaled Dot-Production Attention을 사용한다. 입력 데이터는 d_k의 차원을 가진 query 값들과 key 값들, d_v의 차원을 가진 value값들로 구성된다. 모든 key값들과 query가 내적 연산을 한 후 $\sqrt{d_k}$로 나누어준다. 그런 다음 소프트맥스(softmax)함수를 거쳐 각 value에 대한 가중치 값을 구한다. 한번에 query 값들을 attention 함수에 적용시키기 위해 행렬 Q로 묶고 key 값들과 value 값들 또한 행렬 K와 V로 묶어 연산한다. 행렬 연산의 결과는 다음과 같다.

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

두 개의 가장 많이 사용되는 attention function이 additive attention과 dot-product attention이다. Dot-product attention 여기서 사용한 attention은 $\sqrt{d_k}$로 scaling을 한 것 외에 동일하다. $\sqrt{d_k}$로 scaling을 한 이유는 d_k가 커질수록 소프트맥스(softmax)가 작은 기울기 값(gradient)을 내도록 하기 때문에 이를 보완하기 위한 것이다.

Multi-Head Attention

d_{model}차원의 query, key, value 값들에 한 번의 attention을 적용하는 것 대신 query, key, value값들이 h번 각각 다르게 d_k,d_k, d_v차원으로 선형적으로 사형된(선형 변환) 값을 사용하는 것이 더 좋다는 것을 발견하였다. Query, key, value값들이 사영된(선형 변환) 값들에 대해 병렬적으로 attention 함수를 적용하여 d_v의 차원의 값을 생성하고 h개의 값들을 병합하여 한번 더 선형 변환한 후 최종적으로 값을 출력한다. Multi-head attention은 모델이 다른 위치에 다른 표현(representation) 하위 공간으로부터 오는 정보에 공동으로 집중할 수 있게 한다. (한국어 임베딩; 하나의 문장을 여러 사람이 분석한 것 같다)

Multi-head attention에 대한 식은 다음과 같다.

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

W_i^Q : query 집합 곧 행렬 Q와 연산하는 가중치 행렬 (d_{model} x d_k)

W_i^K : key 집합 곧 행렬 k와 연산하는 가중치 행렬 (d_{model} x d_k)

W_i^V : value 집합 곧 행렬 V와 연산하는 가중치 행렬 (d_{model} x d_v)

W_i^O : 각 head의 출력값들 병합한 값과 연산하는 가중치 행렬 (hd_v x d_{model})

이 논문에서는 h=8 즉 8개의 헤드로 어텐션 층을 적용하였다. 각 어텐션 층에서 각 입력 데이터의 차원은 d_k = d_v = d_{model}/h = 64 이며, 감소된 차원으로 인해 전체 차원의 입력 데이터에 하나의 어텐션을 적용한 것과 비슷한 연산량이 되었다.

Applications of Attention in our Model

Transformer에서는 세 가지 방법으로 Multi-head attention을 적용하였다.

1) Encoder와 decoder가 연결되어 연산이 되는 encoder-decoder attention 층에서 query 데이터는 디코더의 이전 부(sub) 층에서 오고 key, value데이터는 encoder에서 온다. 이는 디코더에서 모든 위치의 데이터가 입력 데이터의 모든 위치에 집중할 수 있도록 한다. 이것은 이전의 sequence to sequence model에서의 encoder-decoder attention의 방법을 따른 것이다. 즉 이전 Seq2seq model에 적용된 attention 알고리즘은 encoder에서의 데이터(hidden state data)와 decoder에서의 데이터를 입력 데이터로 하여 어텐션 알고리즘을 수행한 것과 같다는 것이다.

2) Encoder는 self-attention을 포함한다. Self-attention층에서 사용되는 key, value, query데이터는 모두 같은 곳으로부터 온다. 즉 이전 층의 출력으로부터 key, value, query 데이터가 파생된다.(key, value, query가 같은 값) 이로 인해 encoder에서 데이터의 각 위치에서 이전 층의 출력 데이터의 모든 위치에 집중(attention)할 수 있다.

3) Encoder와 유사하게 self-attention 층이 decoder에도 있다. decoder에서는 각 위치에서 이전 위치에 있는 데이터에 대해 집중(attention)할 수 있도록 한다. 이는 auto-regressive 성질을 보존하기 위해 정보의 흐름이 왼쪽 혹은 역으로 흐르지 않게 한 것이다. 이것을 한 이유는 RNN과 같이 순차적 연산이 없기 때문이라고 생각한다. 이 논문에서 이를 구현하기 위해 올바르지 않은(auto-regressive에 위배되는) 위치에 있는 데이터들은 scaled dot-product attention연산 내부에서 softmax함수에 들어가기 전 -∞값으로 설정함으로써 마스킹하였다.

Position-wise Feed-Forward Networks

Attention 층과 더불어 인코더와 디코더의 각 층마다 fully connected feed-forward 네트워크를 포함한다. 두 번의 선형 변환으로 구성되어 있고 그 사이에 ReLU 활성화 함수가 있다.

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

인코더와 디코더에 선형변환 층이 모두 있지만 각각 다른 Parameter를 사용한다. FFN에서 입력 차원과 출력 차원은 d_{model} = 512이지만 내부 히든 스테이트 데이터의 차원은 d_{ff} = 2048 로 구현하였다.

Positional Encoding

모델에 순환성이나 Convolution 연산이 없기 때문에 모델이 시퀀스 데이터의 순서를 이용하게 하기 위해 시퀀스 데이터 내 각 토큰의 절대 위치 혹은 상대 위치 정보를 모델에 주입하여야 했다. 인코더 층과 디코더 층 밑단에 입력 데이터 임베딩 층에 'Positional Encoding'을 더하였다. Positional Encoding은 임베딩된 값과 같은 차원(d_{model})이기 때문에 두 값이 더해질 수 있다. Positional Encoding에는 여러 방법이 있지만 여기선 주기가 다른 사인 함수와 코사인 함수를 사용하며 아래와 같다.

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

pos값은 position이고 i는 dimension이다. Positional encoding의 각 차원은 사인파와 일치한다. pos는 입력 문장(시퀀스 데이터)에서의 임베딩 벡터의 위치이고 i는 임베딩 벡터 내의 차원의 인덱스를 의미한다. 임베딩 된 벡터 내의 인덱스가 짝수이면(pos, 2i) 사인 함수를 사용하고 홀수이면(pos,2i+1) 코사인 함수를 사용한다. 삼각함수의 파장은 2π에서부터 $10000 * 2\pi$까지 진행되면서 형성한다. 즉 2π에서부터 $10000 * 2\pi$까지가 한번의 주기가 되는 것이다. 이렇게 positional encoding vector를 post마다 구한다면 비록 같은 column이라고 할 지라도 pos가 다르다면 다른 값을 가지게 됩니다. 여기서 이 함수를 선택한 이유는 PE_{pos+k}는 PE_{pos}의 선형 함수의 형태로 표현될 수 있기 때문이다. 이를 보여주기 위해 삼각함수의 덧셈공식을 알아야 한다. 덧셈 공식은 아래와 같다.

$$sin(a+b) = sin(a)cos(b) + cos(a)sin(b)$$
$$cos(a+b) = cos(a)cos(b) - sin(a)sin(b)$$

위 성질을 Positional Encoding에 대한 식에 적용하기 전에 편리를 위해 $c = \frac{1}{10000^{2i/d_{model}}}$이라고 하면

$$PE_{(pos+k,2i)} = sin(\frac{pos+k}{c}) = sin(\frac{pos}{c})cos(\frac{k}{c}) + cos(\frac{pos}{c})sin(\frac{k}{c}) = PE_{(pos,2i)}cos(\frac{k}{c}) + cos(\frac{pos}{c})sin(\frac{k}{c})$$
$$PE_{(pos+k,2i+1)} = cos(\frac{pos+k}{c}) = cos(\frac{pos}{c})cos(\frac{k}{c}) - sin(\frac{pos}{c})sin(\frac{k}{c}) = PE_{(pos,2i+1)}cos(\frac{k}{c}) - sin(\frac{pos}{c})sin(\frac{k}{c})$$

이러한 성질이 모델이 쉽게 상대적인 위치에 따라 attention을 하는 것을 학습할 수 있도록 하기 때문이다.

그리고 학습된 positional embedding방법도 함께 실험했는데 positional Encoding(sinusoidal version)과 positional embedding이 거의 동일한 결과를 냈다. 이 중 sinusoidal version을 사용한 이유는 훈련 시 이 방법이 모델이 훈련 중 학습한 데이터의 시퀀스보다 더 긴 시퀀스 데이터를 만났을 때에도 추론을 가능하게 하기 때문이다.

Why Self-Attention

여기서는 Self-attention 층의 여러 측면과 전형적인 시퀀스 번역 모델에서의 순환 층과 Convolution 연산 층을 비교하고자 한다. Self-attention의 사용 이유로 세 가지가 있다. 첫째는 레이어 당 총 계산 복잡도이다. 두번째는 병렬로 처리될 수 있는 연산량이다.(순차적으로 처리해야 하는 연산이 최소화가 될수록 병렬적으로 연산이 가능한 부분이 많아진다.)

세번째는 네트워크 상에서 장기 의존성이 있는 데이터 상의 거리이다. 장기 의존성이 있는 데이터를 학습하는 것은 많은 번역 태스크에서 핵심적인 도전이었다. 장기 의존성 데이터를 학습하는 능력에 영향을 주는 핵심적인 요소는 전방향, 역방향 신호가 네트워크 상에 지나가야 하는 통로의 길이이다. 인코더와 디코더 사이의 모든 연

결의 길이가 짧을수록 장기 의존성 데이터를 학습하기 쉬워진다. 따라서 다른 계층 유형으로 구성된 네트워크에서 두 개의 입력 위치와 출력 위치 사이의 최대 경로 길이를 비교하고 확인한다. 그래서 네트워크 상 입력 데이터와 출력 데이터의 모든 경로의 길이의 최대를 비교하고자 한다.

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

위 Table1에서, Self-attention층은 순차적으로 연산해야 할 연산을 상수 만큼으로 하면서 모든 위치의 데이터를 연결할 수 있다. 반면에 순환층에서는 O(n) 크기의 연산수가 요구된다. 계산 복잡도 측면에서 시퀀스의 길이(n)가 히든 스테이트 값의 크기(d)보다 작은 경우에 Self-Attention층이 순환층보다 빠르다. 이런 경우는 최신 고성능 모델에서 자주 있는 경우이다. 매우 긴 시퀀스 데이터를 포함한 태스크에서 연산 성능을 개선하기 위해서 Self-Attention층은 출력 시퀀스 데이터에서 각 위치 데이터를 중심으로 입력 데이터에서 참고할 때 오직 r개의 이웃 데이터만 고려할 수 있도록 제한한다. 이는 최대 연결 경로 길이를 O(n/r)으로 증가시킨다. 즉 시퀀스의 길이에 영향을 받게 된다는 것. 이것에 대한 자세한 연구는 추후 있을 연구에서 다룰 예정이다.

커널 크기가 n보다 작은 k인 커널이 있는 convolution층은 입력 데이터와 출력 데이터 상의 연결될 수 있는 경우를 연결하지 않는다. 이는 O(n/k)의 계산 복잡도를 갖는 여러 convolution층이 필요하고 dilated convolution연산이라면 O($\log_k$(n))이 요구된다. 또한 네트워크 상 두 위치의 데이터의 경로의 최대 길이를 증가시킨다. convolution층은 일반적으로 순환층보다 더욱 비용이 크다. 그러나 Separable convolution은 상당히 계산 복잡도를 낮춰준다. 그럼에도 k=n인 경우에는 separable convolution이 self-attention층과 feed forward 층의 결합(Transformer에서 사용하는 구조)과 복잡도가 같아진다.

부가적인 효과로는 self-attention은 더욱 해석가능한 모델을 생성한다. 각각의 attention head는 각각 다른 태스크를 수행하도록 학습할 뿐만 아니라 문장의 문법적, 의미적 구조와 연관되어 수행되어진다.

Training

- Optimizer

  Adam Optimizer를 사용하였고 $\beta_1 = 0.9, \beta_2 = 0.98, \epsilon = 10^{-9}$으로 설정하였다. 훈련 과정에서 학습률이 변하도록 설정하였다. 그에 대한 공식은 다음과 같다.

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

  warmup기법을 사용하였는데 초기 step이 4000동안 학습률이 선형적으로 증가하도록 하였고 그 이후에는 step의 크기의 제곱근의 역수값에 비례하게 감소하도록 설정하였다.

- Regularization

  Residual Dropout 각 층의 출력에서 LayerNorm과 Skip-Connection하기 전에 dropout을 적용하였다. 단어 임베딩값과 positional encoding값의 합에도 dropout을 적용하였다. 기본 모델에서는 Dropout 비율은

0.1을 적용하였다.

- Label smoothing

  훈련 동안 $\epsilon_{ls} = 0.1$의 Label smoothing을 적용하였다.