Abstract

Sequence(문장, 연속적 토큰) 변환(transduction) 모델은 encoder와 decoder를 포함하는 RNN OR CNN으로 구성된다. 이전의 seq2seq with attention도 역시 마찬가지이다.

여기서는 RNN or CNN을 dispense한 attention에만 집중한다.

심지어 parallelizable해서 train time도 줄어들면서 성능도 two translation tasks에 대해 뛰어나다.

28.4 BLEU(WMT 2014 English to German translation task)를 보이고. WMT to French에서는 최고의 성능을 보이며 41.8 BLEU까지 올라간다.(3.5일 동안 8개의 GPU로, 이만큼도 짧다는 것을 논문에서 강조)

1. Introduction

LSTM이나 gated recurrent neural network는 language modeling이랑 machine translation등의 transduction 문제에서 좋은 성능을 보여왔다.

Ht는 ht-1과 현재 input (position t에서)의 영향을 받는다. 하지만 Recurrent 구조는 parallelization을 training 시 불가하게 한다. (input sequence가 길어지면 더 치명적, 메모리도 배치크기에 한계를 가지면서) factorization trick과 conditional computation을 통해 다른 연구는 발전시켰지만 아직 그래도 발전 가능성이 남아있다.

Attention mechanism은 sequence modeling과 transduction에 관련된 많은 tasks에 대해 필수적이다. Input or output sequence에서 거리 관계없이<본 모델에서는 관계를 따진다..는 말인가..?> 모델의 의존성을 키우면서?? 이게뭘까

하지만 attention mechanism은 RNN과 결합해 왔다..

Global dependency를 input과 output사이에서 끌어내며 attention mechanism에만 집중하는 Transformer을 제안한다. <1. Parallelization, 2. Good quality> (어떤 dataset은 12시간 on 8개의 P100 GPUs)

2. Background

과거 CNN을 이용해 sequential computation을 줄이려는 모델이 있었다. (ByteNet, ConvS2S 등) 하지만 input, output사이에서의 연산 횟수가 그 둘의 position사이에서의 거리의 영향으로 많이 증가..? 이게 뭘까(distant postion, 먼 거리는) dependency를 배우기 힘들었다.

Transformer는 constant number of operation을 줄인다. (averaging attention-weighted positions 때문에) 또한 Multi-Head Attention을 설명할 것이다.

Self-Attention은(intra-attention) 한 sequence의 다른 포지션들을 연관시키는 mechanism이다. (두 문장, input output이 아니라?) => sequence를 representation 하기 위한 과정

Reading comprehension, abstractive summarization, textual entailment, task-independent sentence representations에서 사용.

End-to-end memory networks는 recurrent attention mechanism에 집중. 하지만 transformer는 오직 rnn or cnn 없이 input, output의 representation 계산을 위해 self-attention에만 의존

3. model architecture

Encoder는 x1…xn의 symbol representation의 input sequence를 continuous representation sequence, z = (z1…zn)으로 map한다. Z가 주어지면 decoder는 output sequence인 (y1…ym)을 한 symbol씩 한 time마다 생성.

이전에 생성된 symbol을 다음 생성할 때의 추가 input을 사용한다. (model is auto-regressive)

Transformer는 stacked self-attention과 point-wise를 사용한다. Encoder와 decoder에는 둘 다 fully connected layer들이 있다.

3.1. Encoder and Decoder Stacks

Encoder의 전체 stack layer는 N=6 이다. 각 layer는 2개의 sub-layer를 가진다. 첫번째 sub-layer는 multi-head self-attention mechanism이다. 두번째는 simple, position-wise fully connected feed-forward network이다.

Layer normalization에 의해 두 개의 sub layer는 각각 residual connection 된다. 즉, 각 sub-layer는 LayerNorm(x + sublayer(x))이다. 이 과정을 활성화하기 위해서 sub-layer과 embedding layer는 512 dimension의 output을 만들어낸다.

Decoder도 전체 stack layer는 N=6 이다. 2개의 sub-layer외에 3번째 layer가 추가되는데 encoder의 output을 통해 multi-head attention을 수행한다. 역시 normalization layer를 통해 residual connection을 한다.

Self-attention sub-layer를 약간 수정했는데, position이 subsequent positions으로 들어가는 것을 막는다.(masking). Output embedding이 한 position에 의해 offset이라는 사실을 가진 masking은 position i 에서의 prediction은 오직 i 이전의 output들만 알며 의존한다는 것을 보장한다.

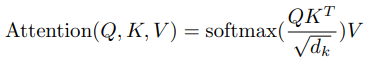
3.2. attention

Attention function은 query, a set of key-value pairs를 output으로 매핑한다. 모든 query, key, value, output은 vector이다. Output은 weighted sum of the values로 계산되고 각 value에 할당된 weight는 상응하는 key와 함께 query의 호환에 의해 계산된다.

3.2.1 scaled dot-product attention

dk의 dimension을 가진 query와 key를 input이 구성한다. 그리고 value의 dimension은 dv이다. 모든 key에 대해 query의 dot products를 계산한다. 그리고 sqrt(dk)로 나눈다. 그리고 softmax를 취해 value의 weight를 얻게된다.

실질적으로 set of query를 동시에 계산한다. (matrix Q를 통해) key와 value들 또한 K, V로 묶인다.



자 위에서 말한거는 일반적인 attention을 따와 우리가 변형한 것인데. 보통 additive attention과 dot-product attention이 있다. 여기에 우리는 sqrt(dk)로 나눈 것뿐이다.

Additive attention은 single hidden layer로 feed-forward network를 이용해 compatibility function을 계산한다.

두 attention은 theoretical complexity에서 유사하지만 dot-product attention이 더 빠르고 공간 효율적이다 (사실상). 따라서 행렬 곱 코드에서 최적화 돼있다.

작은 dk값에 대해서는 두 attention이 비슷하게 작동하지만 additive의 경우 scaling for larger values of dk가 없다면(dk값이 크지 않아 sqrt(dk)로 나눌 필요가 없는 경우…? 이게 뭘까??) 더 잘 작동한다. 하지만 dk값이 클 것으로 예상되어 dot-product값은 매우 커질 것이고 이로 인해 softmax function은 매우 작은 gradients를 가질것이다. 따라서 이것을 막기 위해 dot-product를 scale한다.(sqrt(dk)로)

3.2.2 multi-head attention

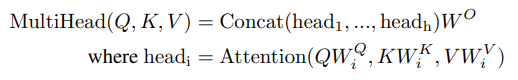
d-model의 dimension을 가지는 key, value, query와 함께 single attention function을 수행하는 것 대신에 다르게, learned linear projection한 dimensions를 h time만큼 project하는 것이 좋다는 것을 발견(dk, dk, dv로)

이렇게 projected version의 query, key, value를 통해 attention을 parallel하게 만들었다.(dv 차원의 dimension을 도출하며)

이것은 다시한번 concatenated and projected 되어 final value를 도출한다.

Multi-head attention은 모델을 결합하여? 동시에? 정보에 참여하게 한다. (다른 포지션에서 다른 representation 부부분공간으로부터)

한 single attention head에서 과정은 다음과 같다.



d-model = 512 / h=8 / dk=dv=64 => 맨 처음 Q와 K와 V는 raw – input matrix WiQ, WiK, WiV는 project 해주는 weight matrix h번 만큼 parallel attention layers를 한 multihead에서 진행 => parallel이라 single-head attention with full dimensionality(single이니까 한 head가 차원은 매우 클것) 과 비슷..

최종적으로 같은 차원의 output 필요 => WO는 h\*dv x d-model 차원

3.2.3 applications of attention in our model

1) decoder에서 query는 이전 decoder layer에서 온다. 메모리에서의 key와 value는 encoder의 output이다.(처음 decode layer 시에??). 이로 인해 디코더의 모든 포지션에서 input sequence의 모든 포지션들에 참여할 수 있다. 기존 인코더-디코더 방식을 모방했다.

2) 인코더는 self-attention layer를 가지고, 모든 key, value, query는 인코더의 이전 레이어의 아웃아웃풋부터 온다. 따라서 인코더의 각 포지션은 이전+현재 포지션에 대해 참여 가능.

3) 디코더에서의 각 포지션도 역시 이전+현재 포지션에 대해 참여 가능하다.

하지만 디코더에서는 Leftward information flow를 막아야한다. (auto-regressive property를 보존하기 위해) 따라서 scaled dot-product attention에서 모든 value를 masking out 한다.(-무한대로) ?? 이게 뭘까

3.3 position-wise feed-forward networks

두 번째 sub-layer로 각 포지션에 대해 분리되어 동등히 적용된다.



레이어 간의 파라미터는 다르지만, 같은 레이어 안에서는 포지션이 달라도 동일한 파라미터를 사용한다. Linear한 transformation의 연산이 부담된다면 convolution으로 사용해도 된다.

Input x와 output은 dimension이 모두 d-model = 512이고 linear transformation 안의 두 레이어는 dff = 2048 dimension이다.

3.4 embeddings and softmax

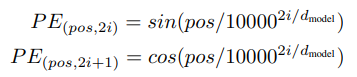
Input, output token들을 d-model dimension의 벡터로 바꾸기 위해 기존에 있단 embedding을 사용한다. 또한 학습된 usual한 linear transformation과 softmax를 사용한다.(decoder ouput을 예측된 다음 토큰의 확률로 바꾸기 위해)

Encoder, decoder에서의 두 embedding layer와 pre-softmax linear transformation에서 같은 weight matrix를 공유하며, embedding layer에서는 sqrt(d-model)을 weight에 곱해준다.

3.5 positional encoding

Sequence의 order를 이용하기 위해 필요하다(rnn이나 cnn을 사용하지 않기 때문에) => token의 relative or absolute position의 정보를 주입한다. => input embedding에 positional encoding을 더함. => embedding의 차원인 d-model = 512와 같으며 따라서 summation 가능.

여러 방법이 있지만, 여기서는 sine, cosine 방식 이용(토큰마다 다른 빈도수에 따라)



인데, pos = position, i = dimension, positional encoding의 각 차원은 sinusoid에 상응.

2파이 ~ 10000\*2파이까지 퍼지며 / 상대적인 포지션에 의해 참가하는 것을 잘 학습할 것이라 예상해서 이것을 선택. Any fixed offset k에 대해 PEpos+k는 PEpos로부터 linear하게 표현 가능.(상대성 이용)

다른 positional embedding을 이용했지만 거의 비슷한 결과 => 하지만 sinusoidal은 모델을 더 긴 length의 sequence extrapolate 가능

4. why self-attention

Self-attention의 세가지 문제를 다룰 것

1) layer당 total computational complexity

2) 최소의 operation이 요구된다 했을 때 paralleizaed로 계산되는 양이다.

3) long-range dependency 사이의 path length 이다.

Long-range dependency를 학습하는 것은 많은 sequence transduction tasks에서 중요하다. Forward and backward로 signals가 왔다갔다 하는 path의 길이가 이 dependency를 배우는 중요 요인이다.

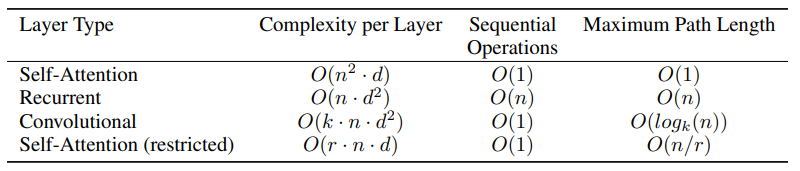
Input, output의 포지션 결합 사이의 짧은 path 일수록 이 dependency를 배우기가 쉽다. 따라서 minimum일 때만 비교하는 것이 아닌 maximum path length일 때의 경우도 따져본다.

Rnn의 경우 O(n)의 sequential operations but self-attention은 모든 포지션을 상수 시간안에 연결시킨다.

Sequence의 길이가 n일 때 n이 representation dimensionality d보다 작다면 self-attention이 빠르다. 이런 case가 대부분의 machine translation의 경우이다.

하지만 n이 매우 큰 long sequence라면 neighborhood of size r로 제한해서 사용한다. (한계점 => 나중에 보완할 것) => 이렇게 되면 모든 input 포지션과 output 포지션을 연결시킬 수 없다.

4. 번 부분은 다시 더 공부 => 너무 복잡



5. training