4. 안녕, 트랜스포머

<블록 시작>

외국에 처음 나가본 윤우는 사람들이 다른 언어로 말하고 있는 것을 처음 보고 깜짝 놀랐다.

윤우: 아빠, 여기 사람들이 뭐라고 하는거에요?

아빠: .... 아빠도 모르겠어...

<블록 끝>

NLP에서 가장 활발하게 연구되고 있는 분야 중 하나가 Neural Machine Translation이다. 기계 번역이라고 한다. 하나의 언어에서 같은 뜻을 가진 다른 언어로 바꾸는 태스크이다. 즉 번역기이다.

2017년 6월에 Attention is all you need라는 논문을 Google에서 발표했다. 이 논문에서 트랜스포머라는 용어가 처음 사용돼었다. 이 논문의 초록을 읽어보면 아래와 같은 몇가지 포인트를 잡을 수 있다.

<블록 시작>

The dominant sequence transduction models are based on complex recurrent or

convolutional neural networks that include an encoder and a decoder. The best

performing models also connect the encoder and decoder through an attention

mechanism. We propose a new simple network architecture, the Transformer,

based solely on attention mechanisms, dispensing with recurrence and convolutions

entirely. Experiments on two machine translation tasks show these models to

be superior in quality while being more parallelizable and requiring significantly

less time to train. Our model achieves 28.4 BLEU on the WMT 2014 Englishto-German translation task, improving over the existing best results, including

ensembles, by over 2 BLEU. On the WMT 2014 English-to-French translation task,

our model establishes a new single-model state-of-the-art BLEU score of 41.8 after

training for 3.5 days on eight GPUs, a small fraction of the training costs of the

best models from the literature. We show that the Transformer generalizes well to

other tasks by applying it successfully to English constituency parsing both with

large and limited training data.

link: https://arxiv.org/pdf/1706.03762.pdf

아래의 부분은 굵은 글씨로

- The dominant sequence transduction models are based on complex recurrent or convolutional neural networks

- We propose a new simple network architecture, the Transformer

- state-of-the-art BLEU score of 41.8 after training for 3.5 days on eight GPUs

인용1

<블록 끝>

[인용1]은 논문 Attention is all you need의 초록이다. 굵은 글씨로된 부분을 보면, 지금까지 대부분의 번역기들은 복잡한 구조의 RNN이나 CNN 구조였는데 이 논문에서는 간단한 구조의 네트워크인 트랜스포머를 제안했고 이는 8개의 GPU에서 3.5일 만에 학습되며 WMT 2014 English-to-French 번역 데이터셋에 대해서 BLEU 스쿠어 41.8이라는 SOTA를 기록했다는 것이다. 트랜스포머는 이후 많은 언어 모델에서 인용되고 있다.

## 주석1: SOTA: 인공지능에서 많은 공식 데이터셋이 있다. 각 데이터셋마다 1등 점수를 state-of-the-art 즉, "예술의 경지"라고 한다.

## BLEU: Appendix 참고

트랜스포머는 RNN이나 CNN을 사용하지 않는다. 자연어의 데이터는 시퀀스이기 때문에 LSTM이나 GRU 등의 순서를 기억하며 학습할 수 있는 모델을 당연하게 사용해왔다. 그러나 이 논문의 저자들은 그 과정이 복잡하다고 판단했고, 그것을 Self-Attention을 통해서 해결했다. Self-Attention울 도입하다보니 GPU를 통해 병렬로 학습할 수 있는 연산이 더 많아졌다. 이 장에서는 트랜스포머가 어떤 구조인지, 왜 좋은 성능을 냈는지, 그리고 구현된 코드까지 공부해보려고 한다.

4.1. 트랜스포머의 구조

이 절에서는 트랜스포머의 구조를 먼저 알아보려고 한다. 트랜스포머의 구조는 크게 인코더와 디코더 부분으로 나뉜다. 인코더는 입력 문장을 은닉 벡터로 표현하는 과정이고 디코더는 은닉 벡터로 표현된 입력 문장을 출력 문장, 즉 번역된 다른 언어의 문장으로 바꾸는 과정이다. 앞 장에서 공부했던 Seq2Seq 모델과 같은 구조이다. 하지만 앞장의 Seq2Seq 모델은 인코더와 디코더에서 RNN 계열의 구조를 사용했고, 트랜스포머에서는 RNN을 사용하지 않고 Self-Attention 구조를 사용한다. Attention 구조에 대해서는 앞 장에서 공부했었다. Self-Attention에 대해서는 앞 장의 내용을 더 확장하여 이번 장에서 다루려고 한다. 트랜스포머에서는 각각 인코더와 디코더가 어떻게 구성되는지 알아보자.

4.1.1. 인코더

트랜스포머의 인코더는 하나의 동일한 블록 구조를 만들어서 그것을 N=6번 반복하는 구조를 하고 있다. 입력 문장을 블록에 넣어서 나온 결과를 다시 또 다른 블록에 넣는 것을 6번 반복해서 트랜스포머의 인코더가 만들어진다. 그 하나의 블록이 어떤 구조를 하고 있는지 알면 트랜스포머 인코더의 구조는 다 공부한 것이다. 트랜스포머의 인코더를 이루는 블록을 자세히 공부해보자.

트랜스포머 인코더의 한 블록은 [그림1]과 같은 구조이다.

<그림 시작>

그림1

<그림 끝>

Multi-head attention 블록과 feed-forward network로 이뤄진다. 먼저 Multi-head attention을 코드로 구현하면 [코드1]과 같다.

<코드 시작>

코드1

class MultiHeadAttention(nn.Module):

'''

MultiHeadAttention with Scaled Dot-Product Attention, default to num\_heads=8

'''

def \_\_init\_\_(self, hidden\_size, num\_heads=8):

super().\_\_init\_\_()

assert hidden\_size % num\_heads == 0

self.num\_heads = num\_heads

self.attention\_head\_size = hidden\_size // num\_heads

self.query = nn.Linear(hidden\_size, hidden\_size)

self.key = nn.Linear(hidden\_size, hidden\_size)

self.value = nn.Linear(hidden\_size, hidden\_size)

self.softmax = nn.Softmax(dim=-1)

self.dropout = nn.Dropout(p=dropout\_rate)

def forward(self, q, k, v, mask=None):

'''

q: Query tensor of shape [batch\_size, maxlen, hidden\_size]

k: Key tensor of shape [batch\_size, maxlen, hidden\_size]

v: Value tensor of shape [batch\_size, maxlen, hidden\_size]

mask: mask tensor of shape [batch\_size, 1, maxlen]

return: Output tensor of shape [batch\_size, maxlen, hidden\_size]

'''

n\_batch = q.size()[0]

split\_shape = (n\_batch, -1, self.num\_heads, self.attention\_head\_size)

if mask is not None:

mask = mask.unsqueeze(1)

# split x for multi-head attention calculation

query = self.query(q).view(\*split\_shape).transpose(1,2)

key = self.key(k).view(\*split\_shape).transpose(1,2)

value = self.value(v).view(\*split\_shape).transpose(1,2)

# calculate scores

scores = torch.matmul(query, key.transpose(-2,-1)) / math.sqrt(self.attention\_head\_size)

# apply mask if mask is not None

if mask is not None:

scores = scores.masked\_fill(mask == 0, -1e9)

attention\_prob = self.softmax(scores)

# make context

context = torch.matmul(attention\_prob, value)

context = self.dropout(context)

# concat multi-head context

merged\_shape = (n\_batch, -1, self.num\_heads\*self.attention\_head\_size)

context = context.transpose(1,2).contiguous().view(\*merged\_shape)

return context

<코드 끝>

[코드1]에서 어텐션을 구할때 Scaled Dot-Product Attention을 사용한다.

<수식 시작>

Scaled Dot-Product Attention 공식

수식1

<수식 끝>

Scaled Dot-Product Attention은 3장에서 다뤘던 어텐션과 비교해서 1/dk가 softmax 전에 추가된 형태를 띈다. 이는 행렬 연산의 결과 값이 너무 커지게 될 경우, 그 값을 softmax할 떄 gradient가 수렴해버리는 현상을 막기 위함이다.

이제 트랜스포머 인코더를 구성하는 두번째 단위 블록인 Point-wise Feedforward Network를 살펴보자.

<코드 시작>

코드2

class PositionWiseFeedForwardNetwork(nn.Module):

'''

Position-Wise Feedforward Network.

입력 텐서의 은닉 사이즈 512를 2048로 확장 후 다시 512로 줄임.

'''

def \_\_init\_\_(self, hidden\_size, inner\_hidden\_size=2048):

super().\_\_init\_\_()

self.w1 = nn.Linear(hidden\_size, inner\_hidden\_size)

self.relu = nn.ReLU()

self.w2 = nn.Linear(inner\_hidden\_size, hidden\_size)

def forward(self, x):

'''

x: Input tensor of shape [batch\_size, maxlen, hidden\_size]

return: Output tensor of shape [batch\_size, maxlen, hidden\_size]

'''

x = self.w1(x)

x = self.relu(x)

x = self.w2(x)

return x

<코드 끝>

Point-wise Feedforward Network는 구현이 간단하다. (hidden\_size x inner\_hidden\_size) weight을 이용해서 행렬 곱을 해준 후 ReLU를 실행한다. 그리고 다시 (inner\_hidden\_size x hidden\_size) weight으로 행렬을 해서 다시 원래의 사이즈로 되돌린다.

트랜스포머의 구현에서 가장 핵심이 되는 단위 블록인 MultiHeadAttention과 PositionWiseFeedForwardNetwork를 구현했다. 이제 이 두 개의 블록을 적절하게 감싸서 트랜스포머를 만들어나가면 된다. 이 절의 제일 첫 부분에서 언급했던 트랜스포머의 전체적인 구조를 다시 한번 언급해보자. 트랜스포머는 크게 인코더와 디코더로 나뉘어 있고, 인코더는 하나의 동일한 구조를 만들어서 N=6번 반복한다고 했다. [코드3]의 구조를 6번 반복하는 것이다.

<코드 시작>

코드3

class EncoderLayer(nn.Module):

'''

EncoderLayer

MultiHeadAttention과 PositionWiseFeedForwardNetwork 두 개의 sublayer로 이뤄져 있다.

위 두 sublayer를 LayerNorm(x + sublayer(x)) 형태로 연산한다.

즉, Residual Network와 Layer Normalization을 추가적으로 수행한다.

'''

def \_\_init\_\_(self, hidden\_size):

super().\_\_init\_\_()

self.self\_attention = MultiHeadAttention(hidden\_size)

self.feedforward = PositionWiseFeedForwardNetwork(hidden\_size)

self.norm = nn.LayerNorm(hidden\_size, eps=1e-6)

def forward(self, x, mask):

'''

x: Input tensor of shape [batch\_size, maxlen, hidden\_size]

mask: mask tensor of shape [batch\_size, 1, maxlen]

return: Output tensor of shape [batch\_size, maxlen, hidden\_size]

'''

x = self.norm(x + self.self\_attention(x, mask=mask))

x = self.norm(x + self.feedforward(x))

return x

<코드 끝>

[코드3]을 보면 LayerNorm(x + sublayer(x)) 형태로 두 서브레이어를 연결해준 것을 제외하고는 [코드1]과 [코드2]에서 만들었던 블록을 그대로 사용하고 있다. (batch\_size, maxlen) 형태의 최초 입력 벡터를 (batch\_size, maxlen, hidden\_size) 형태로 변형해주는 임베딩 레이어와 [코드3]의 EncoderLayer를 6번 반복한 것이 트랜스포머의 인코더이다. 임베딩 레이어는 [코드4]와 같이 구현할 수 있다.

<코드 시작>

코드4

class PositionalEncoding(nn.Module):

'''

PositionalEncoding

짝수번째에 대해서는 sin함수, 홀수번째에 대해서는 cosine함수 값을 정하여 x값에 더해준다.

'''

def \_\_init\_\_(self, hidden\_size, dropout, maxlen=5000):

super(PositionalEncoding, self).\_\_init\_\_()

self.dropout = nn.Dropout(p=dropout)

# Compute the positional encodings once in log space.

pe = torch.zeros(maxlen, hidden\_size)

position = torch.arange(0, maxlen).unsqueeze(1)

div\_term = torch.exp(torch.arange(0, hidden\_size, 2) \*

-(math.log(10000.0) / hidden\_size))

pe[:, 0::2] = torch.sin(position \* div\_term)

pe[:, 1::2] = torch.cos(position \* div\_term)

pe = pe.unsqueeze(0)

self.register\_buffer('pe', pe)

def forward(self, x):

'''

x: Source input tensor of shape [batch\_size, maxlen]

return: Output tensor of shape [batch\_size, maxlen]

'''

x = x + torch.autograd.Variable(self.pe[:, :x.size(1)],

requires\_grad=False)

return self.dropout(x)

class Embeddings(nn.Module):

'''

Embeddings

보통의 Embeddings에 sqrt(hidden\_size)를 더한 값을 리턴한다.

'''

def \_\_init\_\_(self, vocab\_size, hidden\_size):

super(Embeddings, self).\_\_init\_\_()

self.lut = nn.Embedding(vocab\_size, hidden\_size)

self.hidden\_size = hidden\_size

def forward(self, x):

'''

x: Source Input tensor of shape [batch\_size, maxlen]

return: Output tensor of shape [batch\_size, maxlen, hidden\_size]

'''

return self.lut(x) \* math.sqrt(self.hidden\_size)

<코드 끝>

[코드4]까지 구현한 내용들을 반복해서 [코드5]에서 Encoder를 완성할 수 있다. 그런데 뭔가 허전하다. Sequence를 학습하기 위한 무언가가 없다. Transformer 이전에는 그것을 RNN 계열의 구조가 담당했다. Transformer에는 RNN 계열의 구조가 전혀 없다. 시퀀스를 학습할 수 있게 하기 위해서 PositionalEncoding을 더해준 것이다.

<블록 시작>

인용2

Since our model contains no recurrence and no convolution, in order for the model to make use of the

order of the sequence, we must inject some information about the relative or absolute position of the

tokens in the sequence. To this end, we add "positional encodings" to the input embeddings at the

bottoms of the encoder and decoder stacks.

<블록 끝>

[인용2]는 논문 Attention is all you need의 일부이다. 모델로 하여금 시퀀스를 사용할 수 있도록 하기 위해 상대적인 혹은 절대적인 정보를 넣어줘야 한다고 하고 이것을 positional encoding이 하고 있다고 적혀있다. 그 정보를 위해 더해주는 값이 Positional Encoding이고 시퀀스에 더해주는 값은 아래와 같은 공식으로 정해진다.

<수식 시작>

positional encoding sin cos

수식2

<수식 끝>

<코드 시작>

코드5

class Encoder(nn.Module):

'''

Transformer Encoder

하나의 Embedding 레이어와 N=6 개의 EncoderLayer로 구성돼 있다.

리턴할 때는 LayerNormalization 후 리턴한다.

'''

def \_\_init\_\_(self, vocab\_size, hidden\_size, positional\_encoding, n=6):

super().\_\_init\_\_()

#self.embedding = nn.Embedding(vocab\_size, hidden\_size)

self.embedding = nn.Sequential(Embeddings(vocab\_size, hidden\_size), positional\_encoding)

self.layers = nn.ModuleList([EncoderLayer(hidden\_size) for \_ in range(n)])

self.norm = nn.LayerNorm(hidden\_size)

def forward(self, x, mask):

'''

x: Source input tensor of shape [batch\_size, maxlen]

mask: Source mask tensor of shape [batch\_size, 1, maxlen]

return: Output tensor of shape [batch\_size, maxlen, hidden\_size]

Example:

>>> pos\_encoder = PositionalEncoding(hidden\_size, 0.1)

>>> encoder = Encoder(vocab\_size, hidden\_size, pos\_encoder)

>>> inputs = torch.tensor([

>>> [1, 15, 35, 45, 5, 8, 10, 2, 0, 0, 0, 0, 0, 0, 0],

>>> [1, 4, 7, 66, 34, 33, 13, 4, 8, 2, 0, 0, 0, 0, 0],

>>> [1, 25, 5, 6, 77, 24, 8, 5, 9, 27, 2, 0, 0, 0, 0],

>>> [1, 16, 13, 8, 3, 75, 37, 9, 2, 0, 0, 0, 0, 0, 0],

>>> ])

>>> inputs.shape

torch.Size([4, 15])

>>> outputs = encoder(inputs)

>>> outputs.shape

torch.Size([4, 15, 512])

'''

x = self.embedding(x)

for layer in self.layers:

x = layer(x, mask=mask)

return self.norm(x)

<코드 끝>

[코드5]가 트랜스포머의 인코더 전부이다. 가장 기본 단위 블록이었던 MultiHeadAttention과 PositionWiseFeedForwardNetwork를 구현한 다음에는 이렇게 간단하게 구현된다. 인코더를 구현해봤으니 이제 디코더 부분도 구현해보자.

4.1.2. 디코더

[코드1]과 [코드2]에서 구현했던 클래스들을 이용해서 DecoderLayer를 [코드6]과 같이 구현할 수 있다.

<코드 시작>

코드6

class DecoderLayer(nn.Module):

'''

DecoderLayer

EncoderLayer와 같은 블록을 이용하지만 MultiHeadAttention이 한번 더 들어간다.

그 외 Residual Network나 Layer Normalization을 추가하는 것은 같다.

'''

def \_\_init\_\_(self, hidden\_size):

super().\_\_init\_\_()

self.self\_attention\_1 = MultiHeadAttention(hidden\_size)

self.self\_attention\_2 = MultiHeadAttention(hidden\_size)

self.feedforward = PositionWiseFeedForwardNetwork(hidden\_size)

self.norm = nn.LayerNorm(hidden\_size, eps=1e-6)

def forward(self, x, m, mask):

'''

x: Input tensor of shape [batch\_size, maxlen, hidden\_size]

m: Memory tensor of shape [batch\_size, maxlen, hidden\_size] from the Encoder

mask: Mask tensor of shape [batch\_size, 1, maxlen]

return: Output tensor of shape [batch\_size, maxlen, hidden\_size]

'''

x = self.norm(x + self.self\_attention\_1(x, x, x, mask))

x = self.norm(x + self.self\_attention\_2(x, m, m, mask))

x = self.norm(x + self.feedforward(x))

return x

<코드 끝>

Encoder와 마찬가지로 Decoder 역시 DecoderLayer를 6개 반복하는 과정을 통해 만든다. 한가지 다른 점은 MultiHeadAttention 블록을 두번 사용한다는 것이다. 두번째 사용한 MultiHeadAttention 블록이 forward 함수에서 어떻게 사용되는지 보면 차이점을 알 수 있다. Encoder의 MultkHeadAttention에서는 query, key, value에 같은 값이 들어가지만, Decoder의 두번째 MultiHeadAttention에서는 key와 value에 다른 값이 들어간다. 이 값은 Encoder의 아웃풋이다. Decoder 전체 구현체를 보려면 [코드7]을 살펴보자.

<코드 시작>

코드7

class Decoder(nn.Module):

'''

Transformer Decoder

하나의 Embedding 레이어와 N=6 개의 DecoderLayer로 구성돼 있다.

리턴할 때는 LayerNormalization 후 리턴한다.

'''

def \_\_init\_\_(self, vocab\_size, hidden\_size, positional\_encoding, n=6):

super().\_\_init\_\_()

self.embedding = nn.Sequential(Embeddings(vocab\_size, hidden\_size), positional\_encoding)

self.layers = nn.ModuleList([DecoderLayer(hidden\_size) for \_ in range(n)])

self.norm = nn.LayerNorm(hidden\_size)

def forward(self, x, m, mask):

'''

x: Target input tensor of shape [batch\_size, maxlen]

m: Memory tensor of shape [batch\_size, maxlen, hidden\_size] from the Encoder

mask: Target mask tensor of shape [batch\_size, 1, maxlen]

return: Output tensor of shape [batch\_size, maxlen, hidden\_size]

'''

x = self.embedding(x)

for layer in self.layers:

x = layer(x, m, mask=mask)

return self.norm(x)

<코드 끝>

4.1.3. Transformer

Encoder와 Decoder를 만들었으니 이제 두 개를 합쳐서 Transformer 클래스를 구현해보자.

<코드 시작>

코드8

class Transformer(nn.Module):

'''

Transformer

Encoder와 Decoder로 구성된 Transformer 클래스이다.

Positional Encoding에 사용되는 weight은 공유하기 때문에 pos\_encoder를 Encoder와 Decoder 양쪽에서 모두 사용했다.

'''

def \_\_init\_\_(self, source\_vocab\_size, target\_vocab\_size, hidden\_size):

super().\_\_init\_\_()

pos\_encoder = PositionalEncoding(hidden\_size, 0.1)

self.encoder = Encoder(source\_vocab\_size, hidden\_size, pos\_encoder)

self.decoder = Decoder(target\_vocab\_size, hidden\_size, pos\_encoder)

def forward(self, source, target, source\_mask, target\_mask):

'''

source: Source input tensor of shape [batch\_size, maxlen]

target: Target input tensor of shape [batch\_size, maxlen]

source\_mask: Source mask tensor of shape [batch\_size, 1, maxlen]

target\_mask: Target mask tensor of shape [batch\_size, 1, maxlen]

'''

print('\*\* source: {}'.format(source.shape))

print('\*\* target: {}'.format(target.shape))

print('\*\* source\_mask: {}'.format(source\_mask.shape))

print('\*\* target\_mask: {}'.format(target\_mask.shape))

memory = self.encoder(source, source\_mask)

print('\*\* encoder output: {}'.format(memory.shape))

output = self.decoder(target, memory, target\_mask)

print('\*\* decoder output: {}'.format(output.shape))

return output

<코드 끝>

[코드1]부터 [코드8]까지의 구현체를 Attention is all you need의 실제 내용과 비교해보면서 공부해보자. 지금까지 구현한 Transformer는 [코드8]과 같이 정의해서 사용할 수 있다.

<코드 시작>

코드9

>>> source\_vocab\_size = 100

>>> target\_vocab\_size = 110

>>> hidden\_size = 512

>>>

>>> model = Transformer(source\_vocab\_size, target\_vocab\_size, hidden\_size)

>>> source = torch.tensor([

>>> [1, 15, 35, 45, 5, 8, 10, 2, 0, 0, 0, 0, 0, 0, 0],

>>> [1, 4, 7, 66, 34, 33, 13, 4, 8, 2, 0, 0, 0, 0, 0],

>>> [1, 25, 5, 6, 77, 24, 8, 5, 9, 27, 2, 0, 0, 0, 0],

>>> [1, 16, 13, 8, 3, 75, 37, 9, 2, 0, 0, 0, 0, 0, 0],

>>> ])

>>> target = torch.tensor([

>>> [1, 55, 35, 65, 25, 4, 15, 12, 9, 2, 0, 0, 0, 0, 0],

>>> [1, 42, 27, 16, 3, 6, 13, 24, 58, 5, 7, 2, 0, 0, 0],

>>> [1, 75, 2, 34, 79, 96, 3, 55, 19, 7, 2, 0, 0, 0, 0],

>>> [1, 66, 24, 3, 13, 83, 24, 19, 3, 2, 0, 0, 0, 0, 0],

>>> ])

>>> source\_mask = (source != pad).unsqueeze(-2)

>>> target\_mask = (target != pad).unsqueeze(-2)

>>> out = model(source, target, source\_mask, target\_mask)

>>> source.shape, target.shape, source\_mask.shape, target\_mask.shape

(torch.Size([4, 15]),

torch.Size([4, 15]),

torch.Size([4, 1, 15]),

torch.Size([4, 1, 15]))

>>> out.shape

torch.Size([4, 15, 512])

<코드 끝>

4.2. Why Transformer

처음 Transformer 구조를 공부했을 때가 떠오른다. 구현된 모델의 구조를 보는데 RNN 계열의 구조가 보이지 않는 것이다. 시퀀스를 다루는데 있어서 당연하게 포함됐던 RNN 계열의 구조가 없었던 것이다. 당시에 개인적으로 꽤나 충격적이었다. 논문을 읽어보면 Self-Attention을 사용한 이유를 크게 세가지를 들고 있다.

- 레이어 단위별로의 컴퓨팅 연산

- 병렬처리될 수 있는 컴퓨팅의 연산

- 먼 거리에 있는 단어들끼리의 의존성을 학습하기 위함.

첫번째는 각 레이어마다의 기본적인 컴퓨팅 연산이 얼마나 들어가는지에 대한 내용이다. 즉 시간복잡도이다. RNN, CNN계열의 시간복잡도와 Self-Attention의 시간복잡도를 비교하면 [표1]과 같다.

<표 시작>

표1

Self-Attention O(n^2\*d) O(1)

RNN O(n\*d^2) O(n)

CNN O(k\*n\*d^2) O(n)

Self-Attention(restricted) O(r\*n\*d) O(n/r)

<표 끝>

[표1]을 보면 n < d일 경우, Self-Attention이 RNN보다 빠르다는 것을 볼 수 있다. 논문에서는 n이 커질 경우에는 Self-Attention을 할 때 입력의 일부분만을 고려하는 Self-Attention(restricted)를 제안했다. 입력의 일부분을 고려한다는 것은 한 입력의 전체 길이에 해당하는 어텐션을 구하는 것이 아니라 일부 길이(r)에 해당하는 부분에 대해서만 어텐션을 구한다는 것이다. 이럴 경우 시간복잡도가 O(r\*n\*d)로 줄어들게 된다. 대신에 Self-Attention(restricted)를 사용할 경우 단점은 Maximum path length를 찾는 시간복잡도가 O(1)에서 O(n/r)로 늘어나게 된다.

4.3. 트랜스포머 학습 결과

이 절에서는 트랜스포머가 학습된 결과와 성능에 대해서 이야기해보려고 한다. 트랜스포머의 성능을 평가하기 위해 PPL과 BLEU라는 두 가지 평가 지표를 사용한다. 이 절에서 사용한 결과는 논문에 나온 결과이며, 이 책의 소스코드로 실행한 결과가 아니다.

4.3.1. Perplexity(PPL)

우선 PPL에 대해서 알아보자. Perplexity의 약자로 언어 모델의 성능을 나타내는데 사용한다. Perplexity를 영어사전에서 찾아보면 무언가를 이해할 수 없어서 느끼는 당혹감이라고 나와있다. 유사한 단어로 confusion이 있다. 즉 PPL은 얼마나 혼동스러운가를 나타내는 지표로 보면 된다. 따라서 낮을수록 더 좋은 값을 갖게 된다. PPL을 구하는 공식은 [수식3]과 같다.

<수식 시작>

수식3

PPL 구하는 공식

<수식 끝>

수식이 복잡해보이지만 해석해보면 한 시퀀스에 대한 확률 값을 구한 후, 그 값에 -(1/n) 제곱을 해준 결과이다. 예를 들어보자. 주사위를 10번 던지는 경우에 대한 PPL은 아래와 같이 구할 수 있다.

<블록 시작>

1/6 10번 던지는 PPL 구하는 공식

<블록 끝>

PPL을 조금 더 쉽게 설명하면, PPL 값만큼의 경우의 수로 다음 단어를 예측하려고 하는 것으로 이해할 수 있다. 각각의 확률이 1/6인 경우에 대해서 PPL이 6이 나왔다. 즉, 6개 중에 한 개를 다음 입력값으로 고민하고 있다고 해석할 수 있다. 여기에서는 언어 모델의 PPL에 대해서 이야기를 하고 있으니 언어 모델의 PPL도 비교해보도록 하자.

<표 시작>

표2

트랜스포머 이전의 PPL들 https://research.fb.com/building-an-efficient-neural-language-model-over-a-billion-words/

ngram PPL : https://web.stanford.edu/~jurafsky/slp3/3.pdf

트랜스포머의 PPL: https://arxiv.org/pdf/1706.03762.pdf Table3

<표 끝>

[표2]의 PPL은 서로 같은 비교대상으로 구한 PPL은 아니다. 그렇지만 값의 범위 값을 비교해보면 좋을 것 같다. N-gram 모델에서는 PPL이 100단위이다. Unigram의 경우 970까지 나온다. 다음 단어를 예측하는데 후보 단어가 970개나 있다는 뜻이다. 그러면 자연스럽게 다음 단어를 예측하는 확률이 떨어질 것이며 이는 모델이 자연어를 이해하는 성능에 영향을 미칠 것이다. 반면에 AI 기반의 모델은 PPL이 훨씬 낮다. 특히 트랜스포머의 경우 10 이하의 PPL을 보여주고 있다. [표2]가 동일한 조건에서 구한 PPL이 아니라는 점을 다시 한번 강조한다. [표2]를 통해서 언어 모델에 비해서 AI/트랜스포머 언어 모델이 자연어를 이해하는 지표의 수치가 훨씬 높다는 것을 보이해했다면 충분하다.

N-gram 기반의 언어 모델에서는 N-gram 확률을 직접 곱해서 PPL을 구할 수 있다. 그러나 단어와 단어간의 확률을 명시적으로 구할 수 없는 인공지능 모델의 경우 어떻게 엔트로피를 구할 수 있을까? 인공지능 모델의 학습에 많이 사용하는 loss인 Cross Entropy loss를 이용해서 PPL을 구한다. 이 내용은 이 절의 성격과 거리가 있기 때문에 [부록1]에 따로 자세하게 정리해뒀다.

## [부록1] PPL = Cross Entropy Loss??

4.3.2. BLEU 스코어

논문 Attention is all you need에서는 트랜스포머를 이용해서 번역기를 만들었다. 번역 성능을 측정하는 지표로 BLEU가 있다. BLEU 스코어는 Bilingual Evaluation Understudy Score의 약자이다. 사람이 번역한 결과와 기계가 번역한 결과가 얼마나 서로 유사한지를 비교하는 지표이다. 이번 절에서는 BLEU 스코어의 계산법에 대해서 간단하게 언급한 후 트랜스포머의 BLEU 스코어도 언급해보려고 한다.

문장 번역은 사실 정확한 정답이 없다. 하나의 문장을 다른 문장으로 번역했을때 서로 다른 문장이지만 같다고 볼 수 있는 여러가지 경우의 수가 있기 때문이다. 따라서 BLEU 스코어에서는 정답 문장을 여러개 둘 수 있게 디자인돼 있다. "나는 너를 항상 사랑해."라는 문장을 번역기 A, B, C가 각각 영작했다고 해보자.

<블록 시작>

번역할 문장: 알지? 나는 너를 항상 매우 사랑해.

candidateA: know that I always love you that much.

candidateB: You love I always.

candidateC: I I I I I.

정답1: You know, I always love you so much.

정답2: Know right? I always love you a lot.

<블록 끝>

굳이 BLEU 스코어를 계산하지 않아도 candidateA가 가장 좋은 문장이라는 것은 쉽게 알 수 있다. candidateB는 문법이 엉망이다. candidateC는 그냥 엉망이다. 여기에서는 제대로 BLEU 스코어를 구하는 방법을 알아보고 있으니 위의 candidate을 정답1과 정답2를 이용해서 평가해보자. 정답1과 정답2에서 나온 모든 단어를 열거해보면 다음과 같다.

<블록 시작>

know right I always love you so much a lot

<쁠록 끝>

각각의 unigram count를 위 단어들에 대해서 각각 구한 후 candidateA, candidateB, candidateC의 단어 수로 나눠보자.

<블록 시작>

candidateA: (know(1) + I(1) + always(1) + love(1) + you(1) + much(1)) / 8

candidateB: (I(1) + always(1) + love(1) + you(1)) / 4

candidateC: (I(5)) / 5

<쁠록 끝>

여기에서 무언가 이상함을 느껴야 한다. 스코어를 구하고 있으니 스코어는 높어야 좋은 것인데 candidateB(4/4)와 candidateC(5/5)가 candidateA(6/8)보다 더 높다. 왜 그럴까? candidateB의 경우 순서를 고려하지 않았기 때문이고 candidateC의 경우에는 정답1, 정답2에는 한번 밖에 나오지 않았던 단어 I가 5번이나 나왔기 때문이다. candidateB의 경우 N-gram을 통해서 해결할 수 있고, candidateC의 경우 정답 문장에서 발현한 단어의 개수를 참고하는 조금 다른 카운팅 기법을 이용해서 해결할 수 있다. 단어의 실제 카운트 수와 비교해서 더 적은 카운트를 사용하는 것이다. 가령 정답1, 정답2에서 나온 you의 카운트 중 더 큰 수(2)와 candidateA에서 출현한 you의 개수(1)을 비교해서 더 적은 값(1)을 택하는 것이다. 이 방법을 modified unigram precision이라고 한다. unigram 대신에 N-gram을 사용하면 modified N-gram precision이 된다. modified unigram precision을 이용해서 candidate A~C를 다시 계산하면 아래와 같은 결과를 얻을 수 있다.

<블록 시작>

candidate A~C까지의 modified unigram precision

0.3779644730092272 # know that I always love you that much.

0.3423503955179092 # You love I always.

0.3423503955179092 # I I I I I.

<블록 끝>

candidateB의 경우 순서가 엉망이다. 이 순서를 고려하도록 N=1~4로 두고 modified N-gram precision을 수행해보자.

<블록 시작>

# You love I always.

modified 1-gram precision: 4/4

modified 2-gram precision: 1/3

modified 3-gram precision: 0/2

modified 4-gram precision: 0/1

<쁠록 끝>

위 결과는 [코드10]을 실행해서 얻은 결과이다. [코드10]에는 modified N-gram precision을 구하는 함수가 구현돼 있다.

<코드 시작>

코드10

def word\_counter(tokens, n):

return Counter(ngrams(tokens, n))

def modified\_count(candidate, references, n):

cand\_cnt = word\_counter(candidate.split(), n)

temp = Counter()

for ref in references:

tokens = ref.split()

ref\_cnt = word\_counter(tokens, n)

for k, v in ref\_cnt.items():

if k in temp:

temp[k] = max(ref\_cnt[k], temp[k])

else:

temp[k] = ref\_cnt[k]

return Counter({tok:min(temp[tok], cand\_cnt[tok]) for tok in cand\_cnt})

def modified\_ngram\_precision(candidate, references, n):

min\_cnt = modified\_count(candidate, references, n)

total\_cnt = word\_counter(candidate.split(), n)

min\_cnt\_sum = sum(min\_cnt.values())

total\_cnt\_sum = sum(total\_cnt.values())

#print('modified {}-gram precision: {}/{}'.format(n, min\_cnt\_sum, total\_cnt\_sum))

return min\_cnt\_sum / total\_cnt\_sum

>>> candidate = 'You love I always'

>>> references = [

>>> 'You know, I always love you so much',

>>> 'Know right? I always love you a lot',

>>> ]

>>> for i in range(4):

>>> modified\_ngram\_precision(candidate, references, i+1)

<코드 끝>

[코드10]에서는 N-gram을 N=1~4로 두고 실행했다. 각 결과를 보면 unigram일 때는 순서를 해석할 수 없기 때문에 precision이 1.0으로 계산됐지만 bi-gram만 보더라도 precision이 크게 떨어지는 것을 알 수 있다. BLEU 스코어를 구할 때는 1~N까지의 precision에 가중치를 준다. 이 가중치는 정해져있지 않지만 합이 1이 되는 형태로 정의하면 된다.

최종적으로 BLEU 스코어는 [코드11]과 같이 구할 수 있다.

<코드 시작>

코드11

def brevity\_penalty(candidate, references):

cand\_len = len(candidate.split())

diff = np.inf

for ref in references:

tokens = ref.split()

if diff > abs(len(tokens) - cand\_len):

ref\_len = len(tokens)

diff = abs(len(tokens) - cand\_len)

if cand\_len > ref\_len:

return 1

elif cand\_len == 0:

return 0

else:

return np.exp(1 - ref\_len/cand\_len)

def bleu\_score(candidate, reference\_list, weights=[0.25, 0.25, 0.25, 0.25]):

bp = brevity\_penalty(candidate, references)

precisions = [modified\_ngram\_precision(candidate, references, n=i+1) for i in range(len(weights))]

score = 0

for w, p in zip(weights, precisions):

if p == 0:

continue

score += (w \* np.log(p))

return bp \* np.exp(score)

>>> bleu\_score(candidate, references)

0.3423503955179092

<코드 끝>

[코드11]에서 함수 brevity\_penalty에 대해서 간단하게 설명해보면, 짧은 문장은 precision이 높게 나올 가능성이 있기 때문에 그것에 대한 페널티를 주는 것이다. 예를 들어 번역 문장이 "it is"라고 해보자. "it is"는 자주 나올 가능성이 높은 조합이다. 이것만으로 정답 문장들과 precision을 비교해보면 당연히 높게 나올 수 밖에 없다. 그것을 해결하기 위해서 [수식4]와 같은 공식을 이용해서 짧은 문장에 대해서는 페널티를 주는 것이다.

<수식 시작>

수식5

brevity penalty 공식

<수식 끝>

BLEU 스코어를 구하는 공식을 최종적으로 구현했으니 [코드11]을 이용해서 sentence A~C까지의 최종 BLEU 스코어를 구해보면 [표3]와 같은 결과를 얻을 수 있다.

<표 시작>

표3

<표 끝>

트랜스포머의 BLEU 스코어는 EN-FR(영어 → 프랑스어)와 EN-DE(엉어 → 독일어) 모델에서 모두 SOTA를 기록했다.

<표 시작>

표3

논문에 BLEU SOTA 표

Transformer (big) 28.4 41.8

<표 끝>

이번 장에서 트랜스포머가 나오게 된 배경과 트랜스포머의 구조를 알아봤고, 트랜스포머의 학습과 검증 과정에 사용됐던 지표인 PPL과 BLEU에 대해서 알아봤다. Attention is all you need에 나오는 최초의 트랜스포머는 2018년 이후의 급격한 NLP 발전의 모티브로 작용하여 2019년 2020년 다른 논문에서도 많이 인용됐다. 트랜스포머가 Self-Attention을 이용해서 필요한 부분에 대해서만 가중치를 높게 주고 그것을 기반으로 번역문을 생성하여 번역 품질이 급격하게 상승했지만, 그래도 번역기일 뿐이다. 다른 종류의 NLP 테스크를 하기 위해서는 새로운 데이터를 이용해서 학습을 해야하며 이는 많은 양의 데이터와 오랜 시간의 학습 시간을 필요로 한다. 이런 문제를 2018년 10월에 발표된 BERT를 통해 해결 가능하다. 다음 장에서는 BERT에 대해서 알아볼 예정이다.

https://heiwais25.github.io/nlp/2019/10/13/Language-model-3/