3. 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 [인용1]과 같은 몇가지 포인트를 잡을 수 있다.

<인용 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

<인용 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 스코어 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 스코어: 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 내는지 알아보고 트랜스포머를 구현한 소스코드도 살펴보려고 한다.

3.1. 트랜스포머의 구조

트랜스포머의 전체 구조를 알아보자. 이번 절에서는 코드를 사용하지 않고 그림을 통해서만 트랜스포머의 구조를 알아보려고 한다. 어떤 인공지능 모델이든 모델의 입력과 출력을 확실하게 정리해두면 모델의 구조를 이해하기가 훨씬 편해진다. 트랜스포머의 입력은 번역할 문장이고 트랜스포머의 출력은 번역된 문장이다.

<그림 시작>

그림1

<그림 끝>

트랜스포머의 입력은 번역할 문장이고 출력읜 번역된 문장이다. [그림1]에서는 간략하게 표현하기 위해서 입력과 출력을 텍스트로 표현했다. 트랜스포머에서 입력과 출력의 텍스트는 토크나이저를 이용해서 토큰화한 후 각각의 토큰을 숫자로 매핑된다. 토크나이저에 의해서 매핑된 값이 트랜스포머의 입력과 출력이 된다.

<그림 시작>

그림2

<그림 끝>

[그림2]를 보면 토크나이저가 두 개 사용된다. 하나는 번역할 문장인 입력을 토큰화할 입력 토크나이저이고 다른 하나는 번역된 문장을 역토큰화할 출력 토크나이저이다. [그림2]와 같이 토큰화된 입력이 트랜스포머의 입력이 되고, 트랜스포머는 그 입력 값을 이용해서 토큰을 출력한다. 그 출력을 다시 역토큰화해서 번역된 문장을 만든다.

이제 트랜스포머 내부 구조를 들여다보자. 트랜스포머는 크게 인코더와 디코더로 나뉘어있다.

<그림 시작>

그림3

<그림 끝>

[그림3]의 인코더와 디코더는 각각 6개씩의 서브 레이어로 구성돼 있다. 인코더를 구성하는 6개의 서브레이어는 일렬로 연결돼 있다. 일렬로 연결된 서브레이어에 입력 값을 넣으면 첫 서브레이어부터 마지막 서브레이어까지 순서대로 거치게 된다. 디코더는 조금 다르다. 디코더도 인코더와 같이 6개의 서브레이어가 일렬로 구성돼 있지만 각 서브레이어가 받아들이는 입력이 조금 다르다. 디코더의 서브레이어는 이전 서브레이어의 출력과 인코더의 마지막 출력 값을 입력으로 받는다. [그림4]를 참고하라.

<그림 시작>

그림4

<그림 끝>

인코더와 디코더의 서브레이어 구조는 [그림5]를 참고하라.

<그림 시작>

그림5

<그림 끝>

[그림5]를 보면 인코더 서브레이어에 셀프어탠션이 하나 있고, 디코더 서브레이어에는 셀프어탠션과 인코더디코더어탠션이 있다. (##각주1 셀프어탠션은 2장에서 자세하게 설명했었던 어탠션 매커니즘과 같다. 다만 쿼리, 키, 값이 모두 같은 값의 벡터를 갖는다. 쿼리, 키, 값이 모두 같은 값이기 때문에 자기 자신에 대한 어탠션 가중치를 구할 수 있다.) 세 개 모두 멀티헤드 어탠션(Multi-Head Attention) 구조를 하고 있다. 멀티헤드 어탠션의 구조를 먼저 살펴보자.

멀티헤드 어탠션에서 입력 값을 여러 개로 쪼개서 어탠션을 계산한다. 입력 값이 쪼개지는 개수를 N으로 하면 하나의 입력 값을 N개로 쪼개서 각각 어탠션을 계산하게 된다. 결국 어탠션을 N번 하게 되는데 N번이 순차적으로 진행되는 것이 아니라 동시에 병렬적으로 진행된다. GPU 등의 하드웨어를 통해서 병렬로 처리할 수 있다. 즉 멀티헤드 어탠션의 "멀티헤드"는 쪼개진 입력 값이 N번 병렬적으로 실행될 수 있다는 의미를 가지고 있다. [그림6]을 통해서 입력 값이 어떻게 N개로 쪼개지는지 알아보자.

<그림 시작>

그림6

<그림 끝>

[그림6]을 보면 각각의 토큰은 512 사이즈의 벡터로 표현돼 있고, 토큰은 my, name, is, Dooli로 총 네 개이다. 만일 8개의 멀티헤드 어탠션을 하려고 한다면 각각의 512 사이즈 벡터를 8개로 나눠서 64짜리 벡터 8개로 만든다. 그러면 모든 토큰들은 각각 8개의 벡터로 쪼개진다. 2장에서 설명했던 어탠션을 다시 한번 상기해보자. 어탠션의 입력으로는 키, 쿼리, 값이 있다. 키, 쿼리, 값이 모두 [그림6]과 같은 과정을 거치게 되면 [그림7]과 같은 어탠션 연산이 가능하게 된다.

<그림 시작>

그림7

<그림 끝>

[그림7]을 보면 쿼리, 키, 값을 이루는 토큰들이 모두 8개의 64 사이즈 벡터로 쪼개져있다. 그리고 8번의 어탠션 연산이 실행된다. 이 부분에서 일어나는 연산 과정이 [그림5]의 셀프어탠션과 인코더디코더어탠션에서 일어나는 연산이고 트랜스포머의 핵심이다. 중요한 과정이기 때문에 [블록1]을 통해서 텐서 사이즈의 변화를 정리해보도록 하자.

<블록 시작>

블록1

Q = (B,M,H) → (B,M,N,K) → (B,N,M,K)

K = (B,M,H) → (B,M,N,K) → (B,N,M,K)

V = (B,M,H) → (B,M,N,K) → (B,N,M,K)

SCORE = Q x K(T) → (B,N,M,K) x (B,N,K,M) → (B,N,M,M)

PROB = SCORE/루트(K) → (B,N,M,M) → 텐서 사이즈에 변화 없음

ATTN = PROB x V = (B,N,M,M) x (B,N,M,K) → (B,N,M,K)

단,

B = 배치 사이즈

M = 토큰 개수

H = 토큰을 표현하는 벡터 사이즈

N = 멀티해드 개수

K = H / N (항상 H가 나누어 떨어지도록 N을 설정해야함, 즉 K는 항상 정수)

K(T) = 행렬 Transpose 연산. K의 마지막 두 차원의 순서를 바꾼 벡터

ATTN = 어탠션 결과

<블록 끝>

[블록1]과 같은 연산을 통해서 [그림5]의 셀프어탠션과 인코더디코더어탠션을 연산하면 [그림8]과 같은 어탠션 연산 결과를 갖게 된다.

<그림 시작>

그림8

<그림 끝>

트랜스포머 모델의 핵심이 이 부분이다. [그림8]과 같은 입력에 대한 어탠션 연산을 멀티헤드 방식으로 진행하는 것이다. 이 외에도 자세하게 보면 레이어 정규화(Layer Normalization)과정이나 드롭아웃(DropOut) 등의 연산이 중간 중간에 있다. 모델의 성능에 직접적으로 영향을 주는 중요한 부분이지만 입력의 사이즈를 변형시키는 연산은 아니기 때문에 그 부분에 대한 추가적인 설명은 생략하려고 한다.

3.2. 트랜스포머 구현하기

트랜스포머를 간단하게 구현해보자. 이 절에서 구현하는 트랜스포머는 트랜스포머의 모델 구조를 이해하기 위한 샘플이다. 모델 성능에 영향을 주나 입력의 사이즈를 변형시키지 않는 레이어 정규화나 드롭아웃 등의 구현은 생략했고 핵심인 멀티헤드 어탠션 부분을 중점적으로 구현했다.

트랜스포머는 인코더와 디코더 부분으로 나뉘어 있기 때문에 [코드1]과 같이 인코더/디코더 부분을 중점적으로 구현해보자. 구현에 앞서 이 절 전체에서 사용할 기호를 먼저 [코드1]과 같이 변수로 정의했다.

<코드 시작>

코드1

B = 64 # 배치 사이즈

M = 10 # 토큰의 최대 길이

V = 1024 # 토큰의 개수

N = 8 # 멀티헤드 개수

H = 512 # 토큰의 임베딩 사이즈

EXP = 2048 # 확장 사이즈 (FeedForward 클래스 참고)

<코드 끝>

<코드 시작>

코드1

class Transformer(nn.Module):

def \_\_init\_\_(self):

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

self.encoder = Encoder(L)

self.decoder = Decoder(L)

def forward(self, src, dst):

'''

data = np.random.randint(0, V, (B, M))

src = torch.from\_numpy(data)

data = np.random.randint(0, V, (B, M))

dst = torch.from\_numpy(data)

src.shape, dst.shape

m = Transformer()

v = m(src, dst)

v.shape # torch.Size([64, 10, 512])

'''

src\_encoded = self.encoder(src)

dst\_decoded = self.decoder(dst, src\_encoded)

return dst\_decoded

<코드 끝>

[코드1]의 주석을 보면 트랜스포머의 입력은 src(번역할 문장) -> (B, M)과 dst(번역된 문장) -> (B, M)으로 구성된다는 것을 알 수 있다. 트랜스포머의 출력이 (B, M, H)가 되는데 이 사이즈의 벡터는 로그 소프트맥스 함수 등을 통해서 각 (B, M) 사이즈로 변하게 된다. 각 입력들이 인코더와 디코더에서 어떻게 처리되는지 구현을 통해 알아보자.

3.2.1. 인코더

인코더 부분의 구현을 살펴보자.

<코드 시작>

코드2

class Encoder(nn.Module):

def \_\_init\_\_(self, n\_layers):

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

self.n\_layers = n\_layers

self.embedding = Embedding(V, H)

self.layers = [EncoderLayer(H) for i in range(n\_layers)]

def forward(self, x):

'''

data = np.random.randint(0, V, (B, M))

x = torch.from\_numpy(data)

m = Encoder(L)

v = m(x)

v.shape # torch.Size([64, 10, 512])

'''

x = self.embedding(x)

for layer in self.layers:

x = layer(x)

return x

<코드 끝>

[코드2]를 보면 Embedding 레이어에서 H 사이즈만큼으로 임베딩한 후 EncoderLayer를 몇 번 반복해서 돌리는 구조를 띄고 있다. EncoderLayer 내부를 살펴보려면 [코드3]을 참고하라.

<코드 시작>

코드3

class Embedding(nn.Module):

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

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

self.hidden\_size = hidden\_size

self.embedding = nn.Embedding(n\_vocab, hidden\_size)

def forward(self, x):

'''

data = np.random.randint(0, V, (B, M))

x = torch.from\_numpy(data)

m = Embedding(V, H)

v = m(x)

v.shape # torch.Size([64, 10, 512])

'''

return self.embedding(x)

class EncoderLayer(nn.Module):

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

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

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

self.feedforward = FeedForward(hidden\_size, EXP)

def forward(self, x):

'''

x = torch.rand((B, M, H))

m = EncoderLayer(H)

v = m(x)

v.shape # torch.Size([64, 10, 512])

'''

x = self.self\_attention(x, x, x)

x = self.feedforward(x)

return x

<코드 끝>

[코드3]에서는 Embedding과 EncoderLayer를 구현했다. EncoderLayer 부분을 보면 MultiHeadAttention 부분이 있고 이 부분에 대한 자세한 설명은 3.1절에서 자세하게 다뤘다. MultiHeadAttention 레이어에서 멀티해드 개수 만큼의 어탠션 연산이 실행된다. [코드4]를 통해서 어탠션 연산을 수행하는 함수와 MultiHeadAttention 클래스의 내부를 살펴보자.

<코드 시작>

코드4

def attention(query, key, value, scale):

score = torch.matmul(query, key.transpose(-2, -1)) / scale

prob = F.softmax(score, dim=-1)

attn = torch.matmul(prob, value)

return attn

class MultiHeadAttention(nn.Module):

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

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

self.num\_head = num\_head

self.dk = hidden\_size // self.num\_head

def forward(self, query, key, value):

'''

x = torch.rand((B, M, H))

m = MultiHeadAttention(N, H)

v = m(x, x, x)

v.shape # torch.Size([64, 10, 512])

'''

n\_batch = query.shape[0]

query = query.view(n\_batch, -1, self.num\_head, self.dk).transpose(1, 2)

key = key.view(n\_batch, -1, self.num\_head, self.dk).transpose(1, 2)

value = value.view(n\_batch, -1, self.num\_head, self.dk).transpose(1, 2)

x = attention(query, key, value, self.dk)

x = x.transpose(1, 2).contiguous().view(n\_batch, -1, self.dk \* self.num\_head)

return x

<코드 끝>

MultiHeadAttention 클래스의 내부를 보면 query, key, value 값을 만들기 위해 리사이즈(view 함수)와 transpose 함수를 사용한 것을 확인할 수 있다. 이 과정을 통해서 (B, M, H)였던 입력 값이 (B, N, M, dk)로 변형됐고, 이 값과 attention 함수를 이용해서 셀프어탠션을 수행한다. 참고로 query, key, value의 값이 모두 같을 경우 어탠션은 셀프 어탠션을 수행하게 되고 query와 key, value 값이 다르면 일반적인 어탠션을 수행하게 된다.

디코더에서의 구조도 인코더와 비슷하다.

<코드 시작>

코드5

class DecoderLayer(nn.Module):

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

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

self.self\_attention = MultiHeadAttention(n\_head, hidden\_size)

self.encdec\_attention = MultiHeadAttention(n\_head, hidden\_size)

self.feedforward = FeedForward(hidden\_size, 2048)

def forward(self, x, memory):

'''

x = torch.rand((B, M, H))

mem = copy(x)

m = DecoderLayer(N, H)

v = m(x, mem)

v.shape # torch.Size([64, 10, 512])

'''

x = self.self\_attention(x, memory, memory)

return x

class Decoder(nn.Module):

def \_\_init\_\_(self, n\_layers):

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

self.embedding = Embedding(V, H)

self.layers = [DecoderLayer(N, H) for i in range(n\_layers)]

def forward(self, x, memory):

'''

data = np.random.randint(0, V, (B, M))

x = torch.from\_numpy(data)

mem = torch.rand((B, M, H))

m = Decoder(L)

v = m(x, mem)

v.shape # torch.Size([64, 10, 512])

'''

x = self.embedding(x)

for layer in self.layers:

x = layer(x, memory)

return x

<코드 끝>

[코드5]를 보면 디코더의 구조도 DecoderLayer를 몇 번 반복해서 수행하는 형태인 것을 알 수 있다. 다만 DecoderLayer를 보면 MultiHeadAttention 클래스가 두번 정의돼 있는 것을 알 수 있다. 하나는 셀프어탠션을 위함이고 다른 하나는 인코더에서의 값과 어텐션 연산을 하기 위함이다. DecoderLayer의 입력 값 중에서 memory가 인코더로부터 넘어온 인코더의 출력 값이다.

마지막으로 포지셔널 인코딩에 대해서 알아보자. 트랜스포머에서는 RNN과 같은 순환 구조의 구조를 사용하지 않는다. 따라서 토큰 간의 순서 정보를 학습하기 위해서 약간의 추가적인 순서 정보를 넣어준다. 포지셔널 인코딩은 [코드6]과 같이 구현할 수 있다.

<코드 시작>

코드6

class PositionalEncoding(nn.Module):

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

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

pos\_encoding = torch.zeros(M, hidden\_size)

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

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

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

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

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

self.pos\_encoding = pos\_encoding.unsqueeze(0)

def forward(self, x):

'''

x = torch.rand((B, M, H))

m = PositionalEncoding(H)

v = m(x)

v.shape # torch.Size([64, 10, 512])

'''

x = x + Variable(self.pos\_encoding[:, :x.size(1)],

requires\_grad=False)

return x

<코드 끝>

포지셔널 인코딩은 Embedding 레이어를 통해 임베딩된 벡터에 대해서 실행된다. 이 포지셔널 인코딩은 인코더와 디코더의 입력 모두 필요하다. [코드7]은 인코더와 디코더에 포지셔널 인코딩을 추가한 것이다.

<코드 시작>

코드7

class Encoder(nn.Module):

def \_\_init\_\_(self, n\_layers):

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

self.n\_layers = n\_layers

self.embedding = Embedding(V, H)

self.position = PositionalEncoding(H)

self.layers = [EncoderLayer(H) for i in range(n\_layers)]

def forward(self, x):

'''

data = np.random.randint(0, V, (B, M))

x = torch.from\_numpy(data)

m = Encoder(L)

v = m(x)

v.shape # torch.Size([64, 10, 512])

'''

x = self.embedding(x)

x = self.position(x)

for layer in self.layers:

x = layer(x)

return x

class Decoder(nn.Module):

def \_\_init\_\_(self, n\_layers):

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

self.embedding = Embedding(V, H)

self.position = PositionalEncoding(H)

self.layers = [DecoderLayer(N, H) for i in range(n\_layers)]

def forward(self, x, memory):

'''

data = np.random.randint(0, V, (B, M))

x = torch.from\_numpy(data)

mem = torch.rand((B, M, H))

m = Decoder(L)

v = m(x, mem)

v.shape # torch.Size([64, 10, 512])

'''

x = self.embedding(x)

x = self.position(x)

for layer in self.layers:

x = layer(x, memory)

return x

<코드 끝>

3.3. 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 어탠션을 구하는 것이 아니라 일부 길이(r)에 해당하는 부분에 대해서만 어탠션을 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.

3.4. 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.

3.4.1. 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??

3.4.2. 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 [코드8]을 실행해서 얻은 결과이다. [코드8]에는 modified N-gram precision을 구하는 함수가 구현돼 있다.

<코드 시작>

코드8

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>

[코드8]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 [코드9]과 같이 구할 수 있다.

<코드 시작>

코드9

def brevity\_penalty(candidate, references):

candidate = candidate.lower()

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

diff = np.inf

for ref in references:

ref = ref.lower()

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):

candidate = candidate.lower()

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

weights = [1/cand\_len] \* cand\_len

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)

<코드 끝>

[코드9]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 [수식2]와 같은 공식을 이용해서 짧은 문장에 대해서는 페널티를 주는 것이다.

<수식 시작>

수식2

brevity penalty 공식

<수식 끝>

BLEU 스코어를 구하는 공식을 최종적으로 구현했으니 [코드9]을 이용해서 sentence A~C까지의 최종 BLEU 스코어를 구해보면 [표3]와 같은 결과를 얻을 수 있다.

<표 시작>

표3

cand = 'know that I always love you that much.'

bleu\_score(cand, references)

0.6046116297711386

cand = 'you love I always.'

bleu\_score(cand, references)

0.30934850332660563

cand = 'I I I I I.'

bleu\_score(cand, references)

0.5488116360940264

<표 끝>

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

<표 시작>

표4

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.