4. 중간부터 학습하자! 사전학습과 파인튜닝

<블록 시작>

아직 말을 못하는 윤우한테 지난 2년간 윤우 엄마 아빠는 매일 매일 이런 저런 얘기를 해줬다.

드디어 윤우는 조금씩 단어를 말하기 시작하면서 두 개의 단어를 연결해서 말을 하기 시작한다. "맘마 마니~"

아빠의 질문에 대답도 한다. "윤우 잠 얼마나 잤어?" "마니~"

<블록 끝>

아무 것도 모르는 아이들에게 알아듣든 못 알아듣든 엄마 아빠는 계속 말을 해준다. 그러면 자연스럽게 조금씩 언어를 배워나가기 시작한다. 단어 하나를 학습하게 되고 두개를 학습하게 되고, 단어와의 관계를 학습하며 순서도 학습하게 된다. 특정 단어들끼리는 항상 같이 붙어 다닌다는 것을 알게 되며 나아가 문맥간의 논리적인 관계도 이해하게 된다. 컴퓨터로 하여금 언어를 이해하게 하기 위해서 확률 기반의 N-gram 방식을 오랫동안 사용했었고 이를 2장에서 설명했다. 그리고 인공지능 기법을 이용해서 컴퓨터가 언어를 이해하게 하는 방법을 3장에서 설명헀다. 그 다음 4장 트랜스포머에서 번역기를 타겟팅한 획기적인 모델을 소개했다. 이번 장에서 소개할 BERT와 그 이후에 발표된 언어 모델들은 트랜스포머의 구조를 이용해서 만든 언어모델이다.

4.1. 사전학습과 Fine-Tuning

이번 장을 제대로 이해하기 위해 먼저 알아야 할 개념이 사전학습과 파인튜닝이다. 사전학습(Pre-training)의 사전은 미리의 의미를 지닌 사전(事前)이다. Fine-Tuning은 이미 만들어져 있는 것을 조금 변형하는 것이다. 자동차를 튜닝한다고 할 때의 그 튜닝과 같은 뜻이다.

사전학습은 한번만 하면 되며 많은 데이터와 많은 학습시간 또는 자원(GPU 등등)이 요구된다. 파인튜닝은 사전학습된 것을 조금 변형해서 개발자가 최종적으로 원하는 모델로 학습시키는 과정이며, 사전학습 단계에 비해 적은 데이터, 적은 학습시간 또는 자원으로 학습할 수 있다.

예를 들어서 설명해보자. A사에 다니는 홍길동 대리는 영화 감상평을 구분하는 모델을 개발하는 프로젝트를 수행하고 있다. B사에 다니고 있는 둘리 사원은 뉴스 데이터에 대한 질의응답 프로젝트를 수행하고 있다고 하자. 홍길동 대리와 둘리 사원은 서로 다른 회사에서 일하고 있기 때문에 사용하는 데이터의 성격이 매우 다를 것이다. 홍길동 대리는 사용자들이 뎃글에 남긴 영화 감상평을 다루게 될 것이고 둘리 사원은 기자들이 작성한 뉴스 데이터를 다루게 될 것이다. 뿐만 아니라 만드는 모델도 다르다. 홍길동 대리는 텍스트를 분류하는 텍스트 분류 모델을 만들어야 하고 둘리 사원은 질의응답 기능을 가진 모델을 만들어야 한다.

홍길동 대리와 둘리 사원이 가령 BERT를 이용해서 프로젝트를 수행한다면, 둘은 서로 다른 회사에서 서로 다른 성격의 프로젝트를 수행 중이지만 똑같은 사전학습 모델을 사용할 수 있다. 둘이 사용하고 있는 데이터는 서로 다른 성격의 데이터이지만 모두 한국어 데이터이기 때문에 한국어를 사전학습한 한국어 BERT 모델을 공통으로 사용할 수 있다. 그 다음에는 홍길동 대리는 자기 나름대로 BERT 모델에 텍스트 분류를 위한 레이어를 추가해서 영화 감상평 데이터를 사용해 프로젝트를 수행하면 된다. 마찬가지로 둘리 사원은 BERT 모델에 질의응답을 위한 레이어를 추가해서 뉴스 데이터로 프로젝트를 수행하면 된다.

BERT가 나오기 전이라면 홍길동 대리와 둘리 사원은 모델링을 처음부터 끝까지 각자 했어야 했다. 뿐만 아니라 데이터도 더 많이 필요했을 것이다. 충분히 일반화된 모델을 만들려면 충분히 많은 데이터가 있어야 하기 때문이다. BERT를 이용하면 이 두가지 단점을 모두 해결할 수 있다. BERT 사전학습 모델을 로딩해서 사용하게 되기 때문에 기본적인 BERT 구조를 사용하면 된다. BERT 구조 뒤에 각자 다루는 모델의 성격에 따라 레이어를 추가해주면 끝이다. 이 때 추가하는 레이어는 보통 매우 단순한 형태이다. 이렇게 각자의 목적에 맞게 모델을 조금 추가하는 것을 파인튜닝이라고 한다. 사전학습 후 파인튜닝하는 구조를 취하게 되면 사용하는 데이터의 개수도 더 적게 사용할 수 있다. 충분히 일반화된 모델을 만들려면 충분히 많은 데이터가 있어야 하는 것이 기본인데, 사전학습 단계에서 충분히 많은 데이터로 미리 어느 정도 학습시켜뒀기 때문이다.

BERT 이후에 나오는 트랜스포머 기반의 언어 모델은 거의 모두 사전학습 후 파인튜닝하는 구조로 이루어져 있다. 사전학습을 통해서 모델은 언어 자체를 학습하게 된다. 이 때 언어 자체를 학습하면서 일반화도 같이 학습되는 것이다. 그 다음에 레이어를 추가해서 최종적으로 만들고자 하는 모델을 만들면 훨씬 적은 데이터로 일반화된 성능의 모델을 학습할 수 있다. 이것이 이번 절의 핵심이다.

이번 절에서는 사전학습과 파인튜닝이 왜 필요한지, 어떤 장점이 있는지 알아봤다. 다음 절부터는 BERT와 BERT 이후에 발표된 여러 언어 모델들에 대해서 알아볼 것이다.

4.2. BERT

BERT는 2018년 10월에 구글이 발표한 언어 모델이며, 논문 제목은 "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding"이다. BERT는 레이블이 없는 데이터로부터 레이블을 스스로 생성하여 사전학습을 진행해서 만들어진 Pretrained 모델이다. 이 모델을 레이블이 있는 데이터셋으로 Fine-Tuning해서 11개의 자연어 처리 테스크에서 SoTA를 달성했다. 이번 장에서는 BERT의 모델 구조를 먼저 살펴본 후, BERT에서의 사전학습과 파인튜닝에 대해서 자세하게 이야기해려고 한다.

4.2.1. BERT 모델 구조 이해하기

BERT의 모델 구조를 이해해보자. BERT의 모델 구조를 간단하게 설명하면 입력값을 임베딩하는 임베딩 레이어와 그것을 인코딩하는 인코더로 구분된다. 인코더의 구조는 4장에서 설명한 트랜스포머의 인코더와 같은 구조를 하고 있다.

<그림 시작>

그림1

BERT 모델 rough 구조(임베딩+인코더)

<그림 끝>

우선 BERT의 입력 값부터 알아보자.

<코드 시작>

코드1

>>> from transformers import BertTokenizer

>>> tokenizer = BertTokenizer.from\_pretrained('bert-base-uncased')

>>> inp = tokenizer("There is my school and I love this place", return\_tensors="pt")

>>> inp

{'input\_ids': tensor([[ 101, 2045, 2003, 2026, 2082, 1998, 1045, 2293, 2023, 2173, 102]]), 'token\_type\_ids': tensor([[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]]), 'attention\_mask': tensor([[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]])}

>>> itov = {v:k for k, v in tokenizer.vocab.items()}

>>> list(map(lambda x: itov[x], inp['input\_ids'].numpy()[0]))

['[CLS]', 'there', 'is', 'my', 'school', 'and', 'i', 'love', 'this', 'place', '[SEP]']

<코드 끝>

BERT의 입력값은 [코드1]에서 inp에 해당하는 값이다. inp은 input\_ids, token\_type\_ids, attention\_mask를 가지고 있다. input\_ids는 입력된 값을 토큰으로 변환해서 번호를 매긴 것이다. 각 토큰이 번호를 갖고 있다. token\_type\_ids는 토큰의 타입이다. 사실 BERT의 입력으로는 두 개의 시퀀스가 사용된다. [코드1]에서 사용된 There is my school and I love this place는 첫번째 시퀀스이고, 두번째 시퀀스는 [코드1]에서는 생략했다. 두번째 시퀀스를 같이 사용하는 입력 예제는 5.2.5절에서 소개할 예정이다. 첫번째 시퀀스를 토큰 타입 0으로 정의하고 두번째 시퀀스는 토큰 타입 1로 정의한다. [코드1]은 시퀀스가 하나 뿐이기 때문에 token\_type\_ids는 0으로 체워져 있다. attention\_mask는 어텐션을 적용할지 말지 결정하는 값이다. 시퀀스에는 모두 어텐션을 적용하지만 실제 학습에서 뒤에 붙는 [PAD] 토큰에 대해서는 어텐션을 계산할 필요가 없기 때문에 그것을 구분해주기 위해서 사용되는 값이다.

input\_ids, token\_type\_ids, attention\_mask는 필수적으로 사용되는 BERT의 입력값이다. 여기에 추가적으로 position\_ids를 더 넣을 수 있다. position\_ids는 BERT에게 입력 값의 순서 정보를 제공하기 위함으로 단순히 0부터 차례대로 번호를 매긴 값이다.

[그림2]은 BERT의 입력값을 그림으로 설명하고 있다.

<그림 시작>

BERT 입력값 그림

그림2

<그림 끝>

본격적으로 BERT의 구조를 설명해보자. [그림1]를 다시 보면 BERT의 구조는 크게 임베딩과 인코더로 나뉘어 있다. BERT의 입력 값을 처음으로 받는 곳은 임베딩 레이어이다. 임베딩 레이어에서 어떻게 입력 값을 처리하는지 [그림3]를 통해 알아보자.

<그림 시작>

https://github.com/huggingface/transformers/blob/c89180a9de1fc2e98654812fd1c233c3bc6a8d43/src/transformers/models/bert/modeling\_bert.py#L167 참고

그림3

<그림 끝>

[그림3]을 보자. [그림3]에서 사용되는 입력 값은 [그림2]의 input\_ids, token\_type\_ids, position\_ids를 사용하고 있다. [그림3]에서 input\_ids는 임베딩 레이어에 의해서 input\_vector로 바뀌게 된다. 기본 BERT를 기준으로 임베딩 레이어 E1은 토큰을 768 사이즈의 벡터로 변환한다. 따라서 input\_vector는 11개의 서로 다른 768 사이즈 벡터가 된다. token\_type\_ids도 마찬가지로 임베딩 레이어 E2에 의해서 768 사이즈의 token\_vector로 변환된다. position\_ids 역시 임베딩 레이어 E3에 의해서 position\_vector로 변환된다. 이 과정을 코드로 나타내면 [코드2]와 같다.

<코드 시작>

# E1 레이어 정의. 사용하는 가능한 토큰의 전체 수=30522, 임베딩될 벡터의 사이즈=768

word\_embeddings = nn.Embedding(30522, 768, padding\_idx=config.pad\_token\_id)

# E2 레이어 정의. token\_type\_ids에서 나올 수 있는 값의 개수=2, 임베딩될 벡터의 사이즈=768

token\_type\_embeddings = nn.Embedding(2, 768)

# E3 레이어 정의. 입력 값의 최대 길이=512, 임베딩될 벡터의 사이즈=768

position\_embeddings = nn.Embedding(512, 768)

코드2

<코드 끝>

[그림4]를 보면 BERT의 임베딩 레이어 이후에는 인코딩 레이어가 있다. 인코딩 레이어는 self-attention을 통해서 임베딩 레이어의 출력인 768 사이즈의 벡터들을 다른 768 사이즈의 벡터로 변환하는 레이어이다. BERT에서는 인코딩 레이어 12개를 연속으로 이어서 사용한다.

<그림 시작>

그림4

<그림 끝>

[그림4]에서 보면 인코딩 레이어 L1의 입력 값은 임베딩 레이어의 출력값과 같다.([그림3]의 V0부터 V10) 임베딩 레이어의 출력을 인코딩 레이어 L1에 넣었을 때 H1\_0부터 H1\_10까지의 벡터가 출력이 된다. 이 과정을 12번 반복한 것이 BERT의 전체 인코딩 과정이다. 인코딩 레이어 내부는 [그림5]을 통해서 알아보자.

<그림 시작>

그림5

<그림 끝>

self-attention 레이어의 내부에 대한 자세한 내용은 앞서 4장에서 자세하게 설명했으므로 이번 장에서는 간단하게만 짚고 넘어가자. self-attention에서는 입력 값으로 주어진 11개의 벡터들 중에서 서로 유사한 벡터를 찾아서 가중치를 곱해준다. Intermediate과 Output 레이어에는 간단한 FeedForward 네트워크 구조를 하고 있다. Intermediate 레이어에서는 벡터의 크기를 4배 늘렸다가 다시 Output 레이어에서 원래의 크기로 줄인다. self-attention에 대한 자세한 과정을 이해하려면 4장의 4.1.1 인코더의 [코드1]부터 [코드3]까지를 참고하면 된다. 또한 이 장의 5.2.2에서도 BERT의 입력 값과 출력 값의 shape을 정리하고 있으니 그 부분을 참고해도 좋다.

[그림4]와 [그림5]를 통해서 BERT의 인코딩 레이어를 설명했다. [그림5]에서 BERT의 인코딩은 인코딩 레이어 12개를 반복한다고 설명하고 있고, 각 레이어에서 일어나는 연산들에 대해서 [그림5]에서 설명하고 있다. 이번 절에서는 BERT의 입력값으로 어떤 값이 들어가는지 알아봤고, BERT의 임베딩과 인코딩을 공부했다. 이 과정을 복잡하다고 느낄 수 있어서 다음 절에서는 BERT 입력의 모형(shape)에만 집중해서 정리를 했다.

4.2.2. BERT의 입출력 정리

BERT의 입출력을 정리해보자. BERT를 하나 하나 살펴보면 이해해야 할 요소들이 많다. 하지만 shape에 집중해서 구조를 이해하면 BERT는 꽤 간단한 구조를 가지고 있다. 이번 절에서는 shape에만 집중해서 정리를 하기 떄문에 shape을 정의하기 위한 알파벳 기호를 다음과 같이 정의하고 넘어가자.

- B: 배치 사이즈

- M: 입력의 최대 길이

- E: 임베딩 사이즈

- H: 히든 사이즈

- N: BERT 레이어 개수

- S: 어텐션의 히든 사이즈

입력 값의 shape은 (B, M)으로 표기할 수 있다. [그림6]은 입력 값의 형태를 보여주고 있다.

<그림 시작>

그림6

<그림 끝>

(B, M)의 입력을 시작으로 BERT 내부에서는 어떤 구조로 shape이 바뀌는지 알아보자. 이번 절에서 사용하고 있는 BERT의 구조는 HuggingFace에서 구현한 BERT의 구조이다. 우선 임베딩 구조부터 알아보자.

<그림 시작>

그림7

<그림 끝>

임베딩에서 (B, M)의 입력 값을 E 사이즈만큼 임베딩해서 (B, M, E)로 만든다. [그림7]에서 Embedding 레이어를 네모 한 칸으로 표현했다. 입력이 Embedding 레이어를 거치면 (B, M)이었던 입력 값이 (B, M, E)로 바뀐다는 것이다. Embedding 내부에서 발생하는 연산들에 대해서는 앞 절을 참고하면 된다.

임베딩을 거친 다음에 인코딩 과정을 거치게 된다. 인코딩 과정은 [그림8]과 같다.

<그림 시작>

그림8

<그림 끝>

[그림8]을 보면 (B, M, E)가 입력으로 들어가고, Layer를 거치면 (B, M, H)로 바뀌게 된다. BERT에서 E와 H는 같은 값을 갖는다. 즉 Layer는 (B, M, H) 벡터를 입력으로 받아서 같은 크기이지만 다른 값의 (B, M, H) 벡터로 인코딩해서 출력하는 레이어다. 이 과정을 12번 반복하는 것이 BERT의 전부이다. Layer에서 어떤 연산이 이루어지는지는 [그림9]를 통해서 알아보자.

<그림 시작>

그림9

<그림 끝>

[그림9]에서 보면 Layer에서 Attention, Intermediate, Output 레이어를 거치면서 shape이 (B, M, H), (B, M, 4H), (B, M, H)로 바뀌는 것이다. Intermediate는 행렬 곱과 Activation으로 이루어져 있고, Output 레이어는 행렬 곱과 Layer Normalization으로 이루어져 있다. Attention 레이어는 조금 더 복잡한 구조를 띄고 있다. [그림10]을 통해서 Attention 레이어를 보자.

<그림 시작>

그림10

<그림 끝>

Attention 레이어는 SelfAttention과 SelfOutput으로 이루어져 있다. SelfOutput은 행렬 곱과 Layer Normalization으로 이루어져 있고, SelfAttention은 [그림11]을 보고 이해해보자.

<그림 시작>

그림11

<그림 끝>

SelfAttention 레이어는 [그림11]과 같이 조금 복잡한 구조를 하고 있다. (B, M, E) 사이즈의 입력에 서로 다른 행렬을 곱해서 query, key, value로 만든다. query, key, value의 shape은 (B, M, H)이다. 여기에서 H를 (N, S) 사이즈로 쪼개고 행렬의 순서를 바꾸면(transpose) (B, N, M, S)의 shape을 만들 수 있다. 이 과정을 코드로 나타내면 [코드3]과 같다.

<코드 시작>

def transpose\_for\_scores(x):

old\_size = x.size()

new\_x\_shape = x.size()[:-1] + (4, 5)

x = x.view(\*new\_x\_shape) # 쪼개기

print('new\_x\_shape: {} -> {} -> {}'.format(old\_size, new\_x\_shape, x.permute(0,2,1,3).size()))

return x.permute(0, 2, 1, 3) # Transpose

코드3

<코드 끝>

[코드3]을 실행하면 다음과 같은 결과를 얻을 수 있다.

<블록 시작>

# [1, 3, 20] -> [1, 3, 4, 5] -> [1, 4, 3, 5]

>>> x = torch.from\_numpy(np.array(

[[

[201,202,203,204,205,206,207,208,209,210,211,212,213,214,215,216,217,218,219,220],

[221,222,223,224,225,226,227,228,229,230,231,232,233,234,235,236,237,238,239,240],

[241,242,243,244,245,246,247,248,249,250,251,252,253,254,255,256,257,258,259,260],

],

[

[101,102,103,104,105,106,107,108,109,110,111,112,113,114,115,116,117,118,119,120],

[121,122,123,124,125,126,127,128,129,130,131,132,133,134,135,136,137,138,139,140],

[141,142,143,144,145,146,147,148,149,150,151,152,153,154,155,156,157,158,159,160],

]],

))

>>> tx = transpose\_for\_scores(x)

>>> tx.shape

new\_x\_shape: torch.Size([2, 3, 20]) -> torch.Size([2, 3, 4, 5]) -> torch.Size([2, 4, 3, 5])

>>> tx[0]

tensor([[[201, 202, 203, 204, 205],

[221, 222, 223, 224, 225],

[241, 242, 243, 244, 245]],

[[206, 207, 208, 209, 210],

[226, 227, 228, 229, 230],

[246, 247, 248, 249, 250]],

[[211, 212, 213, 214, 215],

[231, 232, 233, 234, 235],

[251, 252, 253, 254, 255]],

[[216, 217, 218, 219, 220],

[236, 237, 238, 239, 240],

[256, 257, 258, 259, 260]]])

>>> x[0]

tensor([[201, 202, 203, 204, 205, 206, 207, 208, 209, 210, 211, 212, 213, 214, 215, 216, 217, 218, 219, 220],

[221, 222, 223, 224, 225, 226, 227, 228, 229, 230, 231, 232, 233, 234, 235, 236, 237, 238, 239, 240],

[241, 242, 243, 244, 245, 246, 247, 248, 249, 250, 251, 252, 253, 254, 255, 256, 257, 258, 259, 260]])

<블록 끝>

위 결과에서 x[0]의 데이터 값 배열이 어떻게 변해서 tx[0]이 됐는지 보자. 단순히 세로로 나눈 다음에 stack처럼 쌓은 것 뿐이다.

지금까지 설명한 내용이 BERT 모델 구조의 전부이다. 이 구조를 한번에 이해하기는 조금 어려울 수 있다. 조금 더 익숙해지려면 4장과 5.2절을 반복해서 공부하기를 추천한다. 또한 HuggingFace와 같은 오픈소스 등을 활용해서 BERT가 어떻게 구현됐는지 이해해보고, 각 레이어에 어떤 값이 들어가서 어떤 연산이 실행되는지 이해해보면 BERT를 조금 더 쉽게 이해할 수 있을 것이다. 다음 절에서는 사전학습과 파인튜닝을 이해해보자.

4.2.3. 사전학습 이해하기

4.1절에서 사전학습에 대해서 설명했다. 다시 한번 간략하게 설명하면 사전학습은 언어를 학습하는 과정이다. 문장을 분류하거나 문장을 번역하는 모델을 처음부터 학습(from scratch)하는 것보다 언어를 미리 학습해둔 모델에서(사전학습된 모델) 학습을 하면 더 빠르게 학습되기 때문에 사전학습을 하는 것이다. BERT는 사전 학습된 언어모델이다. BERT의 구조는 4장에서 소개한 트랜스포머의 인코더 구조를 띄고 있는데 이 구조에 데이터를 넣어서 미리 학습시킨 것이 사전학습된 BERT이고, 이 사전학습된 BERT는 이미 Google이나 HuggingFace, SKT 등등의 큰 기업 또는 오픈소스 프로젝트에서 많은 컴퓨팅 리소스를 이용해서 학습해뒀고, 학습된 모델을 다운로드 받아서 활용할 수 있다.

BERT가 이전의 모델에 비해 높은 성능을 보일 수 있었던 이유는 BERT의 사전학습이다. 그러면 어떻게 BERT를 사전학습했을까? 직접 학습을 해보기는 힘들어도 원리를 이해해볼 수는 있다. BERT의 사전학습은 언어에 대한 사전학습이다. 따라서 언어를 학습할 때 어떤 식으로 학습하는지를 먼저 생각해보자. Masked Language Model(MLM)과 Next Sentence Prediction(NSP)를 통해 학습을 한다. BERT는 MLM과 NSP의 Loss를 낮추는 방식으로 학습된다. 그러면 MLM과 NSP가 각각 어떤 학습을 하는지 다음 절에서 자세하게 알아보자.

4.2.4. Masked Language Model (MLM)

MLM은 쉽게 말해서 빈 칸 체우기이다. 토큰화되어 들어오는 BERT의 입력을 임의로 마스킹하고 마스킹된 값들을 맞추는 멀티 레이블 분류 문제이다. 즉 입력 토큰의 일부를 임의로 몇 개 선택해서 레이블을 스스로 만든 후에 그 레이블을 다시 맞추는 분류 문제이다. 실제로 사람이 언어를 학습할 때 문제집에서 빈 칸 체우기 문제를 풀면서 학습을 한다. 그 방법을 컴퓨터에게 적용한 것이라고 생각하면 된다. 아래의 예시를 보자.

<블록 시작>

입력 토큰: ['[CLS]', 'there', 'is', 'my', 'school', 'and', 'i', 'love', 'this', 'place', '[SEP]']

마스킹된 토큰: ['[CLS]', 'there', 'is', '[MASK]', 'school', 'and', 'i', '[MASK]', 'this', '[MASK]', '[SEP]']

레이블: [-100, -100, -100, 2026, -100, -100, -100, 2293, -100, 2173, -100]

<블록 끝>

my, love, place가 임의로 선택됐으니 이 토큰들을 마스킹한 후 다시 원래의 토큰인 2026, 2293, 2173으로 맞추는 분류 문제인 것이다. MLM 학습은 이렇게 스스로 레이블을 만들어서 학습하게 되므로 self-supervised learning이라고 한다. MLM을 조금 더 자세하게 알기 위해서 BERT 논문에서 MLM을 설명하는 부분을 살펴보자.

<블록 시작>

인용1

link: https://arxiv.org/pdf/1810.04805.pdf

The training data generator

chooses 15% of the token positions at random for

prediction. If the i-th token is chosen, we replace

the i-th token with (1) the [MASK] token 80% of

the time (2) a random token 10% of the time (3)

the unchanged i-th token 10% of the time. Then,

Ti will be used to predict the original token with

cross entropy loss.

<블록 끝>

[인용1]을 보면 입력 토큰의 15 퍼센트에 해당하는 토큰을 임의로 정한 후, 선택된 15% 중 80%는 [MASK] 토큰으로 바꾸고 10%는 랜덤 토큰으로 변경하고 나머지 10%는 바꾸지 않는다고 한다. 원래의 정답이 분명하게 있는데 왜 굳이 일부는 바꾸고 일부는 바꾸지 않고 등등의 복잡한 과정을 거치는 것일까? 다시 한번 말하자면, MLM은 빈 칸 체우기 문제이다. 빈 칸 체우기 문제인데 [MASK] 토큰에 대해서만 정답을 맞춘다. 최초에 선택된 15%에 대해서만 예측을 하고 나머지 85%는 무시된다. 만일 [인용1]과 같은 마스킹 전략을 거치지 않는다면 모델은 [MASK]된 것에 대해서만 빈 칸을 맞출 줄 아는 모델로 학습돼 버리고 만다. 그런데 [MASK]는 Fine-Tuning과정에서는 절대 나오지 않는 토큰이고 MLM의 목적은 빈 칸 체우기를 통해 문맥을 학습하는 것이다. 따라서 [MASK]라고 선택된 토큰들 중에서 10%는 랜덤한 단어로 치환하고 또 다른 10%는 원래의 입력 토큰을 그대로 가져가는 전략을 사용하는 것이다. 이렇게 하면 [MASK]로 돼 있지 않는 부분에 대해서도 MLM은 예측을 수행할 수 있게 돼서 언어의 문맥을 학습할 수 있게 된다.

MLM에 대한 예시를 아래의 예시를 통해 확인해보자.

<블록 시작>

input ids: [101, 2045, 2003, 2026, 2082, 1998, 1045, 2293, 2023, 2173, 102]

input token: ['[CLS]', 'there', 'is', 'my', 'school', 'and', 'i', 'love', 'this', 'place', '[SEP]']

마스킹된 token: ['[CLS]', 'there', 'is', '[MASK]', 'school', 'and', 'i', '[MASK]', 'this', '[MASK]', '[SEP]']

마스킹 전략 후의 token: ['[CLS]', 'there', 'is', '[MASK]', 'school', 'and', 'i', 'go', 'this', 'place', '[SEP]']

마스킹 전략 후의 input\_ids: [101, 2045, 2003, 103, 2082, 1998, 1045, 2293, 2023, 2173, 102]

※ [MASK]: 103

※ go: 2175

※ place: 2173

label: [-100, -100, -100, 2026, -100, -100, -100, 2293, -100, 2173, -100]

※ my: 2026

※ love: 2293

※ place: 2173

<블록 끝>

위의 예시에서 "input\_token"에서 임의로 my, love, place 세 토큰을 골랐다고 하자. 그리고 my는 [MASK]로, love는 go로 place는 그대로 place로 둬서 마스킹 전략을 적용시키자. 원래 대로라면 [인용1]에서 소개한 마스킹 전략과 같이 80%/10%/10%의 비율을 유지해야 하지만 여기에서는 편의상 비율은 무시하도록 하자. 위의 예시의 "마스킹 전략 후의 token"을 보라. MLM을 수행해야 하는 토큰은 총 3개이다. 4번째 자리에 있는 [MASK]를 my로 예측해야 하고, 8번째 자리에 있는 go를 love로 예측해야 하고, 10번째 자리에 있는 place는 그대로 place로 예측하면 된다. 예측할 때 처음에 [MASK]로 선택되지 않았던 85%에 해당하는 부분에 대해서는 예측을 수행하지 않는다.

복잡하게 느껴질 수 있어서 MLM 과정을 순서대로 정리해봤다.

<블록 시작>

STEP1: 사전 학습된 토크나이저를 이용해서 문장을 토큰화해서 숫자로 바꾼다.

STEP2: STEP1에서 숫자로 바뀐 토큰들 중 15%를 랜덤으로 선택한다.

STEP3: label을 만들어둔다. label을 만들 때는 STEP2에서 선택되지 않은 85%는 -100으로 두고 선택된 15%는 그대로 둔다.

STEP4: STEP2에서 선택된 15% 중 80%는 [MASK] 토큰인 103으로 변경하고, 10%는 임의의 다른 숫자로 변경하고, 나머지 10%는 그대로 둔다.

STEP3에서 만든 것이 MLM의 label이고 STEP4에서 만든 것이 MLM의 입력 값이다.

STEP5: STEP3과 STEP4에서 만든 값을 이용해서 MLM loss를 구한다. Loss를 구할 때 label에서 -100으로 돼 있는 부분에 대해서는 loss를 계산하지 않도록 무시해준다.

<블록 끝>

4.2.5. Next Sentence Prediction (NSP)

MLM이 빈 칸 체우기를 학습하는 것이라면 NSP는 다음 문장 맞추기를 학습하는 과정이다. 두 문장이 주어졌을 때 그 두 문장이 앞/뒤 문장 관계인지(is\_next=1) 그렇지 않으면 그냥 랜덤한 문장인지를(is\_next=0) 학습한다. NSP는 두 문장이 is\_next인지 아닌지를 판단하는 문제이기 때문에 입력 데이터 자체가 두 문장으로 이루어진 쌍으로 돼 있다. 이 두 문장을 sentence\_a와 sentence\_b라고 하자. sentence\_a와 sentence\_b의 앞뒤에는 예약어 토큰을 추가해줘야 한다. 예약어는 아래와 같이 추가해준다.

<블록 시작>

[CLS] sentence\_a [SEP] sentence\_b [SEP]

<쁠록 끝>

위의 예시에서 sentence\_a의 앞뒤에는 [CLS]와 [SEP]이 추가됐다. sentence\_a가 처음 시작이니 그 앞에 [CLS]를 붙여주는 것이고 sentence\_a가 끝난 시점에 또 다시 [SEP]을 붙여준 것이다. 그리고 sentence\_b를 추가해주고 끝나는 지점에 한번 더 [SEP]을 추가해서 입력 데이터를 만드는 것이다. 이렇게 만들어진 입력 데이터에 대해서 sentence\_a와 sentence\_b의 관계에 따라 is\_next=0 또는 is\_next=1이 될 수 있다.

<블록 시작>

# is\_next=0

sentence\_a: I love this candy

sentence\_b: I read a good news

input: ['[CLS]', 'i', 'love', 'this', 'candy', '[SEP]', 'i', 'read', 'a', 'good', 'news', '[SEP]']

label: 0

# is\_next=1

sentence\_a: I love this candy

sentence\_b: Because it is so sweet

input: ['[CLS]', 'i', 'love', 'this', 'candy', '[SEP]', 'because', 'it', 'is', 'so', 'sweet', '[SEP]']

label: 1

<블록 끝>

4.2.6. 사전학습을 위한 데이터셋 준비와 Self-supervised learning

아마 MLM과 NSP를 공부하면서 "도대체 그럼 데이터셋은 저런 어떻게 구했지?"라는 생각이 많이 들었을 것이다. BERT 논문을 보면 사전학습에 사용되는 데이터를 어떻게 준비했는지에 대한 내용이 나와있다. BookCorpus(8억 단어)와 English Wikipedia(25억 단어) 데이터셋을 사용했다고 한다. 중요한 점은 문장 수준의 데이터셋보다는 문서 수준의 데이터셋이 필요하다는 것이다.

<인용 시작>

인용2

It is critical to use a document-level corpus rather than a

shuffled sentence-level corpus such as the Billion

Word Benchmark (Chelba et al., 2013) in order to

extract long contiguous sequences.

https://arxiv.org/pdf/1810.04805.pdf

<인용 끝>

[인용2]는 BERT 논문의 일부이다. 길게 이어져있는 시퀀스를 얻기 위해 문서 수준의 데이터셋이 필요하다고 한다. 이전 절에서 설명헀던 NSP를 봐도 두 문장이 연속적인 문장인지 아닌지를 판단하는 문제를 풀기 때문에 이런 데이터셋을 확보하기 위해서는 문장과 문장이 서로 독립인 문장 단위의 데이터셋 보다는 문장 간의 논리적인 인과관계를 갖고 있는 문서 수준의 데이터셋이 더욱 효과적이다.

문서 수준의 데이터셋을 확보하면 MLM과 NSP에 사용되는 데이터셋을 만들어서 사용할 수 있다. 데이터셋을 토큰화해서 랜덤으로 일부를 마스킹한 후 마스킹 전략을 취하면 MLM에 사용되는 데이터셋이 만들어지는 것이고, 문서 수준의 데이터셋을 문장 단위로 쪼갠 후 순서를 고려해서 문장 쌍을 만들면 NSP에 사용되는 데이터셋을 만들 수 있다. 이렇게 하면 레이블링이 없는 데이터를 이용해서 스스로 레이블링을 만들어서 학습할 수 있는 형태가 된다. 스스로 레이블링을 만들어서 학습한다고 하여 이것을 self-supervised learning이라고 한다.

self-supervised learning의 장점은 데이터에 레이블링이 필요하지 않다는 것이다. 데이터만 있으면 레이블링을 생성할 수 있기 때문이다. MLM의 경우에도 주어진 문장을 토큰화해서 랜덤으로 15% 추출하고 추출한 것에서 각각 80/10/10%씩 마스킹 전략을 취해주는 과정은 충분히 코딩으로 구현할 수 있는 내용이다. MLM을 코딩으로 구현하면서 주어진 입력 데이터에 대해서 레이블링도 코딩으로 만들어낼 수 있다. NSP 역시 마찬가지이다. 가령 10문장이 있을때 핸덤으로 두 쌍을 골랐는데 1번 문장과 2번문장이 골라졌다면 is\_next=1로 되도록 코딩을 하는 것이다. 이렇게 코딩하면 스스로 레이블링을 만들어내는 샘이 된다.

4.2.7. 사전학습 파해치기

앞 절에서 BERT가 사전학습으로 MLM과 NSP를 학습한다는 것을 공부했고 각각은 self-supervised learning이기 떄문에 데이터의 레이블을 스스로 구할 수 있다는 것까지 알았다. 이번 절에서는 실제로 사전학습을 할 때 입력되는 데이터를 살펴보고 그 데이터가 어떻게 레이블링 되는지 살펴보려고 한다.

사전학습을 위해 HuggingFace에서 공개한 BERT 구현체를 사용하려고 한다. HuggingFace는 2016년에 설립된 스타트업이고 언어 모델을 빌드하고 학습하고 실행시킬 수 있는 오픈소스를 제공한다. 이 오픈소스의 이름이 Transformers이다. Transformers를 이용하면 BERT나 BERT 이외의 NLP 모델들을 사전학습부터 Fine-Tuning까지 간단하게 학습할 수 있다. 사전학습을 실행시키기 위해서는 아래와 같이 스크립트를 구동시키면 된다.

<블록 시작>

# 도커 형태로 바꿔서 데이터셋 포함시켜두기 (transformers==4.1.0.dev0)

$ git clone https://github.com/huggingface/transformers.git

$ cd transformers

$ mkdir datasets

$ wget -O datasets/train.txt https://ocw.mit.edu/ans7870/6/6.006/s08/lecturenotes/files/t8.shakespeare.txt

$ cp datasets/train.txt datasets/eval.txt

$ cd examples/legacy

$ python run\_language\_modeling.py --model\_type=bert --output\_dir=outputs --tokenizer\_name=bert-base-uncased --mlm --block\_size=32 --train\_data\_file=../../datasets/train.txt --eval\_data\_file=../../datasets/eval.txt --do\_train --no\_cuda

<블록 끝>

위와 같이 run\_language\_modeling.py를 구동시키면 BERT 사전학습을 실행시킬 수 있다. 위의 학습에서 사용하는 train.txt는 셰익스피어 소설이고 예시를 위한 목적으로 사용됐다. 실제로 BERT를 학습시키기 위해서는 매우 큰 데이터셋이 필요하다. 위와 같이 스크립트를 구동해서 학습 데이터의 형태를 보면 [표1]과 같이 요약할 수 있다.

<표 시작>

표1

BERT 학습 데이터 표로 만든것

<표 끝>

[표1]을 보면 "eyes", "wonder", ".", "not", "mine", "," 토큰들이 랜덤으로 선택된 것을 볼 수 있다. 이렇게 선택된 것 이 외의 토큰들은 모두 -100 치환해서 labels를 만들었다. 그리고 선택된 토큰들 중에서 "eyes", "wonder", "not", "," 토큰들은 [MASK]로 치환됐다. "mine" 토큰의 경우 "1843" 토큰으로 랜덤하게 치환됐고, "." 토큰은 치환하지 않고 그대로 뒀다. BERT의 사전학습은 [표1]의 inputs으로부터 labels를 맞추는 학습을 진행하는데 labels의 값이 -100인 부분에 대해서는 학습을 하지 않는다.

run\_language\_modeling.py에서는 NSP 학습은 수행하지 않았다. MLM에 비해서 NSP의 학습 효과가 떨어지기 때문에 MLM만 학습한 것이다.

4.2.8. 사전학습 정리하기

앞에서 여러가지 예제를 통해서 BERT의 사전학습에 대해서 공부했다. 정리를 하는 차원에서 이번 절에서는 BERT의 사전학습을 핵심만 요약해보려고 한다.

<블록 시작>

1. BERT의 사전학습은 MLM과 NSP를 통해서 수행한다.

2. MLM은 "빈칸 체우기", NSP는 "다음 문장인지 맞추기"이다.

3. MLM과 NSP를 위한 데이터셋은 텍스트 데이터만 있으면 self-supervised learning을 통해 스스로 준비된다.

4. MLM의 경우 하나의 입력 데이터에서 임의로 선택된 15%의 토큰에 대해서만 학습을 수행한다.

5. 4에서 선택된 토큰들 중 80%는 [MASK]로 치환, 10%는 임의의 토큰으로 치환, 나머지 10%는 치환을 하지 않는다.

<블록 끝>

BERT의 사전학습 과정을 이해하는 것은 매우 중요하다. 단순히 BERT를 이해하기 위해서가 아니라 BERT 이후의 다른 모델들의 사전학습도 비슷한 과정이기 떄문이다.

4.2.9. Fine-Tuning 이해하기

앞 절에서 BERT를 어떻게 사전학습하는지 알아봤다. 이번 절에서는 파인튜닝하는 것에 대해서 알아보려고 한다. 5.1 절에서 소개했듯이, 파인튜닝은 사전학습된 모델 구조에 특정 목적을 위한 레이어를 추가하는 것이다. 예를 들면 BERT를 이용해서 텍스트 분류 모델을 만들 수도 있고, 질의응답 모델을 만들 수도 있다. 즉 텍스트 분류 모델을 만들고 싶으면 BERT 모델 구조 이후에 텍스트 분류 모델을 위한 레이어를 추가하면 된다는 것이다. 5.1 절을 다시 한번 읽어보면 홍길동 대리와 둘리 사원 모두 사전학습된 BERT를 사용하고 있다. 사전학습된 BERT에 홍길동 대리는 텍스트 분류 모델을 추가한 것이고 둘리 사원은 질의응답 레이어를 추가하는 것이다. 이것이 파인 튜닝이다.

이번 절에서는 BERT를 파인튜닝해서 텍스트 분류 모델과 질의응답 모델을 만들어보자. 모델을 만들때 공통적으로 구현해야 하는 데이터셋/데이터로더를 구현하는 부분에 대한 설명은 생략하고 학습에 사용되는 데이터셋과 모델구조 그리고 추론 예시를 중점적으로 설명하려고 한다.

4.2.10. 텍스트 분류 모델로 파인튜닝하기

BERT를 이용해서 텍스트 분류 모델을 만들어보자. 오픈 데이터셋 중에서 텍스트 분류를 위한 오픈 데이터셋이 많다. 그 중에서 이번 절에서는 CoLA(Corpus of Linguistic Acceptability) 데이터셋을 사용해보려고 한다. CoLA 데이터셋은 영어 문장 데이터셋이고, 각 문장이 언어학적으로 받아들여지는지(Acceptable=1) 그렇지 않은지(Unacceptable=0) 구분해둔 데이터셋이다.

<표 시작>

CoLA 데이터셋 예시

Label sentence

0 In which way is Sandy very anxious to see if the students will be able to solve the homework problem?

1 The book was written by John.

0 Books were sent to each other by the students.

1 She voted for herself.

1 I saw that gas can explode.

https://nyu-mll.github.io/CoLA/

<표 끝>

이 책의 레포지토리에서 "research/cola\_classification/classifier.ipynb" 주피터 노트북에 텍스트 분류를 위한 모델을 구현했다. 택스트 분류를 위한 데이터는 [코드2]와 같이 다운로드하면 된다.

<코드 시작>

코드2

$ cd research/cola\_classification

# 데이터셋 백업

$ wget https://nyu-mll.github.io/CoLA/cola\_public\_1.1.zip

$ unzip cola\_public\_1.1.zip

$ mv cola\_public ../../data/cola\_classification

<코드 끝>

데이터를 로딩하거나 학습을 위해 배치 단위로 생성하는 부분에 대한 설명은 생략하고 모델의 구조 부분을 살펴보자. 모델 구조는 BertForSequenceClassification 클래스를 사용했는데 이 클래스는 transformers 라이브러리에 있는 클래스이며, BERT를 이용해서 텍스트 분류를 할 때 사용하면 된다. BertForSequenceClassification의 일부를 살펴보면 아래와 같다.

<코드 시작>

코드3

https://github.com/huggingface/transformers/blob/802ffaff0da0a7d28b0fef85b44de5c66f717a4b/src/transformers/models/bert/modeling\_bert.py#L1462

class BertForSequenceClassification(BertPreTrainedModel):

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

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

self.num\_labels = config.num\_labels

self.config = config

self.bert = BertModel(config) # BERT 모델을 정의

self.dropout = nn.Dropout(config.hidden\_dropout\_prob)

self.classifier = nn.Linear(config.hidden\_size, config.num\_labels) # 파인튜닝 레이어를 정의

self.init\_weights()

def forward(

self,

input\_ids=None,

attention\_mask=None,

token\_type\_ids=None,

position\_ids=None,

head\_mask=None,

inputs\_embeds=None,

labels=None,

output\_attentions=None,

output\_hidden\_states=None,

return\_dict=None,

):

r"""

labels (:obj:`torch.LongTensor` of shape :obj:`(batch\_size,)`, `optional`):

Labels for computing the sequence classification/regression loss. Indices should be in :obj:`[0, ...,

config.num\_labels - 1]`. If :obj:`config.num\_labels == 1` a regression loss is computed (Mean-Square loss),

If :obj:`config.num\_labels > 1` a classification loss is computed (Cross-Entropy).

"""

return\_dict = return\_dict if return\_dict is not None else self.config.use\_return\_dict

outputs = self.bert(

input\_ids,

attention\_mask=attention\_mask,

token\_type\_ids=token\_type\_ids,

position\_ids=position\_ids,

head\_mask=head\_mask,

inputs\_embeds=inputs\_embeds,

output\_attentions=output\_attentions,

output\_hidden\_states=output\_hidden\_states,

return\_dict=return\_dict,

)

pooled\_output = outputs[1]

pooled\_output = self.dropout(pooled\_output)

logits = self.classifier(pooled\_output)

...

<코드 끝>

[코드3]을 보면 클래스 정의하는 부분에서 BERT 모델과 파인튜닝 레이어를 정의하는 부분을 찾아볼 수 있다. 그리고 forward 함수에서 BERT의 출력을 이용해서 분류 모델에 사용될 출력을 만드는 과정을 확인할 수 있다. 입력 데이터를 self.bert에 넣어서 그 결과를 outputs에 저장하는 부분이 BERT를 사용하는 부분이다. outputs은 길이가 2인 튜플으로 돼 있다. outputs[0]은 입력 데이터를 토큰 별로 인코딩한 값이다. 따라서 입력 데이터의 shape이 (1, 7)일 경우 outputs[0]의 shape은 (1, 7, 768)이다. 각 토큰을 768 사이즈의 벡터로 인코딩한 것이다. outputs[1]은 outputs[0]을 Pooling한 결과이다. Pooling은 큰 차원의 벡터를 작은 차원의 벡터로 요약하는 것으로 이해하면 된다. 즉 outputs[0]은 입력 데이터를 토큰 별로 인코딩한 값이 들어있고, outputs[1]은 outputs[0]을 768 사이즈의 벡터로 요약한 것이라고 할 수 있다. [코드3]에서 outputs[1]을 pooled\_output에 저장하고 정규화를 위해 드롭아웃 레이어를 거친 후 파인튜닝 레이어인 self.classifier의 입력으로 넣는 코드를 확인할 수 있다. outputs[0]을 Pooling하는 코드는 [코드4]를 참고하면 된다.

<코드 시작>

https://github.com/huggingface/transformers/blob/802ffaff0da0a7d28b0fef85b44de5c66f717a4b/src/transformers/models/bert/modeling\_bert.py#L610

코드4

class BertPooler(nn.Module):

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

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

self.dense = nn.Linear(config.hidden\_size, config.hidden\_size)

self.activation = nn.Tanh()

def forward(self, hidden\_states):

# We "pool" the model by simply taking the hidden state corresponding

# to the first token.

first\_token\_tensor = hidden\_states[:, 0]

pooled\_output = self.dense(first\_token\_tensor)

pooled\_output = self.activation(pooled\_output)

return pooled\_output

<코드 끝>

BertForSequenceClassification 모델을 이용해서 CoLA 데이터셋을 학습시킨 부분이 [코드5]이다. 전체 코드는 "research/cola\_classification/classifier.ipynb"에서 살펴보자.

<블록 시작>

코드5

model = BertForSequenceClassification.from\_pretrained("bert-base-uncased", num\_labels = 2).cuda()

model.train()

for i in range(n\_epoch):

train(model, train\_dataloader, optimizer)

Average Loss = 0.4864): 100%|██████████| 268/268 [00:29<00:00, 9.08it/s]

Average Loss = 0.2898): 100%|██████████| 268/268 [00:29<00:00, 9.09it/s]

Average Loss = 0.1645): 100%|██████████| 268/268 [00:29<00:00, 8.96it/s]

Average Loss = 0.1110): 100%|██████████| 268/268 [00:30<00:00, 8.92it/s]

Average Loss = 0.0855): 100%|██████████| 268/268 [00:30<00:00, 8.73it/s]

Average Loss = 0.0650): 100%|██████████| 268/268 [00:30<00:00, 8.86it/s]

Average Loss = 0.0481): 100%|██████████| 268/268 [00:29<00:00, 8.97it/s]

Average Loss = 0.0405): 100%|██████████| 268/268 [00:29<00:00, 8.98it/s]

Average Loss = 0.0348): 100%|██████████| 268/268 [00:30<00:00, 8.93it/s]

Average Loss = 0.0275): 100%|██████████| 268/268 [00:30<00:00, 8.82it/s]

Average Loss = 0.0335): 100%|██████████| 268/268 [00:30<00:00, 8.89it/s]

Average Loss = 0.0314): 100%|██████████| 268/268 [00:30<00:00, 8.81it/s]

Average Loss = 0.0245): 100%|██████████| 268/268 [00:30<00:00, 8.76it/s]

Average Loss = 0.0253): 100%|██████████| 268/268 [00:30<00:00, 8.80it/s]

Average Loss = 0.0194): 100%|██████████| 268/268 [00:30<00:00, 8.76it/s]

Average Loss = 0.0266): 100%|██████████| 268/268 [00:30<00:00, 8.72it/s]

Average Loss = 0.0220): 100%|██████████| 268/268 [00:30<00:00, 8.72it/s]

Average Loss = 0.0187): 100%|██████████| 268/268 [00:32<00:00, 8.16it/s]

Average Loss = 0.0135): 100%|██████████| 268/268 [00:30<00:00, 8.75it/s]

Average Loss = 0.0230): 100%|██████████| 268/268 [00:30<00:00, 8.92it/s]

<블록 끝>

이 주피터 파일에서 학습 모델을 실행시키는데 필요한 자세한 과정은 생략하려고 한다. 설명이 필요한 부분에 대해서는 주피터 파일에 주석을 남겨놨다.

주피터 파일을 통해 학습이 끝나면 경우 cola\_model.bin이 생성된다. 이 모델을 로딩해서 추론을 하면 아래와 같은 결과를 얻을 수 있다.

<코드 시작>

코드6

>>> labels, preds = inference(model, test\_dataloader)

>>> confusion\_matrix(labels, preds)

array([[ 97, 65],

[ 26, 328]])

<코드 끝>

[코드6]는 추론 결과를 Confusion matrix로 표현한 결과이다. 그런데 위 결과만 가지고는 CoLA 데이터셋에 대해서 얼마나 잘 만들어진 모델인지 알기 어렵다. 벤치마킹을 위해 GLUE 리더보드를 확인해보자. GLUE는 여러 가지 종류의 자연어 처리 테스크를 모아놓은 데이터셋의 집합이다. GLUE에는 CoLA를 비롯한 다양한 데이터셋이 포함돼 있다. 리더보드는 성능을 높은 순서대로 나열한 전광판이다. 따라서 GLUE 데이터셋의 리더보드를 보면 CoLA 데이터셋에 대해서 지금까지 최고의 성능을 낸 모델이 무엇인지 그리고 점수는 몇 점인지 알 수 있다.

<그림 시작>

그림12

https://gluebenchmark.com/leaderboard

<그림 끝>

GLUE 데이터셋에 대한 평가 방법으로 Matthew's Correlation을 사용한다고 한다. Matthew's Corr는 sklearn 패키지에 구현돼 있다.

<코드 시작>

코드7

>>> from sklearn.metrics import matthews\_corrcoef

>>> matthews\_corrcoef(labels, preds)

0.5721810924354415

<코드 끝>

[코드7]을 보면 BERT를 이용해서 만든 모델의 추론 성능은 0.5721 정도로 나온다. 이 값을 [그림2]에서의 벤치마크 결과와 비교하면 약 30위권 정도이다. BertForSequenceClassification 모델에서는 BERT를 로딩한 것 외에는 특별히 최적화를 하지 않았다는 점과 BERT를 능가하는 모델이 BERT 이후에 많이 발표됐다는 점을 고려하면 시간대비 높은 성능의 모델을 만들었다고 생각할 수 있다.

사전학습된 BERT를 사용하지 않고 처음부터 학습하면 어떨까? 사전학습을 하지 않은 BERT로 학습을 할 경우 아래와 같이 Loss가 느리게 떨어지는 것을 확인할 수 있다. 사전학습된 BERT를 사용하지 않고 학습한 모델은 cola\_model\_no\_pretrained.bin으로 저장했다. 모델 로딩할 때 파일명을 변경해서 로딩 후 테스트하면 된다. [코드8]은 사전학습된 BERT를 사용하지 않고 학습한 결과이다.

<블록 시작>

코드8

config = BertConfig.from\_pretrained('bert-base-uncased')

model = BertForSequenceClassification(config).cuda()

model.train()

for i in range(n\_epoch):

train(model, train\_dataloader, optimizer)

Average Loss = 0.6211): 100%|██████████| 268/268 [00:31<00:00, 8.41it/s]

Average Loss = 0.6150): 100%|██████████| 268/268 [00:31<00:00, 8.38it/s]

Average Loss = 0.6123): 100%|██████████| 268/268 [00:32<00:00, 8.24it/s]

Average Loss = 0.6136): 100%|██████████| 268/268 [00:31<00:00, 8.43it/s]

Average Loss = 0.6140): 100%|██████████| 268/268 [00:32<00:00, 8.20it/s]

Average Loss = 0.6139): 100%|██████████| 268/268 [00:32<00:00, 8.20it/s]

Average Loss = 0.6112): 100%|██████████| 268/268 [00:32<00:00, 8.19it/s]

Average Loss = 0.6038): 100%|██████████| 268/268 [00:33<00:00, 8.12it/s]

Average Loss = 0.5606): 100%|██████████| 268/268 [00:32<00:00, 8.17it/s]

Average Loss = 0.5141): 100%|██████████| 268/268 [00:32<00:00, 8.19it/s]

Average Loss = 0.4693): 100%|██████████| 268/268 [00:32<00:00, 8.14it/s]

Average Loss = 0.4246): 100%|██████████| 268/268 [00:32<00:00, 8.20it/s]

Average Loss = 0.3828): 100%|██████████| 268/268 [00:32<00:00, 8.33it/s]

Average Loss = 0.3571): 100%|██████████| 268/268 [00:31<00:00, 8.60it/s]

Average Loss = 0.3187): 100%|██████████| 268/268 [00:29<00:00, 8.98it/s]

Average Loss = 0.2886): 100%|██████████| 268/268 [00:29<00:00, 9.00it/s]

Average Loss = 0.2751): 100%|██████████| 268/268 [00:29<00:00, 9.04it/s]

Average Loss = 0.2675): 100%|██████████| 268/268 [00:29<00:00, 8.95it/s]

Average Loss = 0.2352): 100%|██████████| 268/268 [00:29<00:00, 9.03it/s]

Average Loss = 0.2171): 100%|██████████| 268/268 [00:29<00:00, 9.05it/s]

<볼륵 끝>

cola\_model\_no\_pretrained.bin을 로딩해서 추론을 하면 [코드9]와 같은 결과를 얻을 수 있다. [코드7]에서 얻었던 스코어 0.5721보다 훨씬 낮아진 것을 알 수 있다.

<코드 시작>

코드9

>>> labels, preds = inference(model, test\_dataloader)

>>> confusion\_matrix(labels, preds)

array([[ 31, 131],

[ 56, 298]])

>>> from sklearn.metrics import matthews\_corrcoef

>>> matthews\_corrcoef(labels, preds)

0.04111147909054526

<코드 끝>

4.2.11. 질의응답 모델로 파인튜닝하기

BERT를 이용해서 질의응답 모델을 만들어보자. 앞 절에서는 BERT를 이용해서 분류 모델을 만들었다. 이번 절에서는 질의응답 데이터셋으로 유명한 SQuAD(Stanford Question Answering Dataset) 데이터셋을 이용해서 질의응답 모델을 만들어보자. SQuAD 데이터셋은 1.1버전과 2.0버전이 있는데, 이 책에서는 2.0 버전을 사용하려고 한다. 우선 SQuAD 데이터셋의 예시를 살펴보자.

<블록 시작>

"paragraphs":

[

{

"qas":

[

{

"question": "In what country is Normandy located?",

"id": "56ddde6b9a695914005b9628",

"answers": [

{"text": "France", "answer\_start": 159}, {"text": "France", "answer\_start": 159},

{"text": "France", "answer\_start": 159}, {"text": "France", "answer\_start": 159}

],

"is\_impossible": false

},

{

"question": "When were the Normans in Normandy?",

"id": "56ddde6b9a695914005b9629",

"answers": [

{"text": "10th and 11th centuries", "answer\_start": 94},

{"text": "in the 10th and 11th centuries", "answer\_start": 87},

{"text": "10th and 11th centuries", "answer\_start": 94},

{"text": "10th and 11th centuries", "answer\_start": 94}

],

"is\_impossible": false

},

]

"context": "The Normans (Norman: Nourmands; French: Normands; Latin: Normanni) were the people who in the 10th and 11th centuries gave their name to Normandy, a region in France. They were descended from Norse (\"Norman\" comes from \"Norseman\") raiders and pirates from Denmark, Iceland and Norway who, under their leader Rollo, agreed to swear fealty to King Charles III of West Francia. Through generations of assimilation and mixing with the native Frankish and Roman-Gaulish populations, their descendants would gradually merge with the Carolingian-based cultures of West Francia. The distinct cultural and ethnic identity of the Normans emerged initially in the first half of the 10th century, and it continued to evolve over the succeeding centuries."

}

...

]

<블록 끝>

하나의 지문(context)에 대해서 여러 개의 질의응답 쌍(qas)을 갖고 있는 구조이다. qas는 여러 개의 질의응답 쌍이고, 각 쌍은 id, question, answers, is\_impossible 등의 키를 가지고 있다. id는 질의응답 쌍에 대한 고유 키이다. question은 평문으로 된 질문이다. 이 질문은 context로부터 만들어진 질문이다. answers는 질문에 대한 응답이고, text와 answer\_start로 이루어져 있다. 예를 들어서 In what country is Normandy located? 라는 질문에 대해서 answer는 France(text)이고 이는 context의 159번째 문자(answer\_start)이다.

SQuAD 데이터셋을 이용해서 질의응답 모델을 만들때 모델의 입출력은 아래와 같다.

<블록 시작>

# 평문으로 된 question/text

question = "Who was Jim Henson?"

text = "Jim Henson was a nice puppet"

# question과 text를 구분자([CLS], [SEP])로 구분하여 아래와 같이 inputs을 만든다

inputs = ['[CLS]',

'who',

'was',

'jim',

'henson',

'?',

'[SEP]',

'jim',

'henson',

'was',

'a',

'nice',

'puppet',

'[SEP]']

# 위 입력에 대한 정답인 start\_position과 end\_position을 인덱스로 나타낸다.

start\_position = [10]

end\_position = [12]

<블록 끝>

inputs는 question과 text를 토큰화 하여 [CLS], [SEP]등으로 구분해서 연결한 것이다. 질의응답 모델의 입력은 inputs를 숫자로 변환한 값이며 이 때 inputs에 대한 token\_type\_ids와 attention\_mask 값도 같이 입력된다.

<코드 시작>

코드10

from transformers import BertTokenizer, BertForQuestionAnswering

model = BertForQuestionAnswering.from\_pretrained('bert-base-uncased')

input\_ids = [[101, 2040, 2001, 3958, 27227, 1029, 102, 3958, 27227, 2001, 1037, 3835, 13997, 102]]

token\_type\_ids = [[0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1]]

attention\_mask = [[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]]

inputs = {

'input\_ids':torch.tensor(input\_ids),

'token\_type\_ids':torch.tensor(token\_type\_ids),

'attention\_mask':torch.tensor(attention\_mask)

}

start\_position = torch.tensor([[10]])

end\_position = torch.tensor([[12]])

<코드 끝>

BertForQuestionAnswering의 구조는 [코드11]과 같다. 클래스를 정의하는 \_\_init\_\_ 부분을 보면 BERT 모델과 파인튜닝 레이어가 전부이다. [코드11]에는 입력 데이터를 self.bert에 넣어서 그 결과를 outputs에 저장하는 부분이 있다. 여기에서의 outputs은 길이가 1인 튜플이다. outputs[0]은 입력 데이터가 (1, 14)일 경우 (1, 14, 768)이다. outputs[0]을 self.qa\_outputs의 입력으로 넣으면 (1, 14, 2) shape의 텐서가 logits에 저장된다. 이 logits을 split 함수를 통해서 두 개의 (1, 14, 1) 텐서로 쪼갠 후 각각을 squeeze하여 start\_logits과 end\_logits을 만든다. 이 값을 [코드10]에서 준비한 start\_position, end\_position과 대조해서 start\_loss와 end\_loss를 구한다. BertForQuestionAnswering의 forward 함수를 보면 각 단계별로 중요한 부분에 주석으로 텐서의 shape 또는 value를 남겨뒀다.

<코드 시작>

코드11

https://huggingface.co/transformers/\_modules/transformers/models/bert/modeling\_bert.html#BertForQuestionAnswering -> 일부 내용을 삭제/추가했습니다.

class BertForQuestionAnswering(BertPreTrainedModel):

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

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

self.num\_labels = config.num\_labels

self.bert = BertModel(config, add\_pooling\_layer=False)

self.qa\_outputs = nn.Linear(config.hidden\_size, config.num\_labels)

self.init\_weights()

def forward(

self,

input\_ids=None,

attention\_mask=None,

token\_type\_ids=None,

position\_ids=None,

head\_mask=None,

inputs\_embeds=None,

start\_positions=None,

end\_positions=None,

output\_attentions=None,

output\_hidden\_states=None,

return\_dict=None,

):

r"""

example:

- input\_ids: [[101, 2040, 2001, 3958, 27227, 1029, 102, 3958, 27227, 2001, 1037, 3835, 13997, 102]]

- token\_type\_ids: [[0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1]]

- attention\_mask: [[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]]

"""

return\_dict = return\_dict if return\_dict is not None else self.config.use\_return\_dict

outputs = self.bert(

input\_ids, # shape=torch.Size([1, 14])

attention\_mask=attention\_mask, # shape=torch.Size([1, 14])

token\_type\_ids=token\_type\_ids, # shape=torch.Size([1, 14])

position\_ids=position\_ids,

head\_mask=head\_mask,

inputs\_embeds=inputs\_embeds,

output\_attentions=output\_attentions,

output\_hidden\_states=output\_hidden\_states,

return\_dict=return\_dict,

)

sequence\_output = outputs[0] # shape=torch.Size([1, 14, 768])

logits = self.qa\_outputs(sequence\_output) # shape=torch.Size([1, 14, 2])

start\_logits, end\_logits = logits.split(1, dim=-1) # shape=torch.Size([1, 14, 1]), torch.Size([1, 14, 1])

start\_logits = start\_logits.squeeze(-1).contiguous() # shape=torch.Size([1, 14])

end\_logits = end\_logits.squeeze(-1).contiguous() # shape=torch.Size([1, 14])

total\_loss = None

if start\_positions is not None and end\_positions is not None:

# If we are on multi-GPU, split add a dimension

if len(start\_positions.size()) > 1:

start\_positions = start\_positions.squeeze(-1)

if len(end\_positions.size()) > 1:

end\_positions = end\_positions.squeeze(-1)

# sometimes the start/end positions are outside our model inputs, we ignore these terms

ignored\_index = start\_logits.size(1)

start\_positions = start\_positions.clamp(0, ignored\_index)

end\_positions = end\_positions.clamp(0, ignored\_index)

loss\_fct = CrossEntropyLoss(ignore\_index=ignored\_index)

start\_loss = loss\_fct(start\_logits, start\_positions) # value=tensor(2.7861, grad\_fn=<NllLossBackward>)

end\_loss = loss\_fct(end\_logits, end\_positions) # value=tensor(3.1949, grad\_fn=<NllLossBackward>)

total\_loss = (start\_loss + end\_loss) / 2 # value=tensor(2.9905, grad\_fn=<DivBackward0>)

if not return\_dict:

output = (start\_logits, end\_logits) + outputs[2:]

return ((total\_loss,) + output) if total\_loss is not None else output

return QuestionAnsweringModelOutput(

loss=total\_loss,

start\_logits=start\_logits,

end\_logits=end\_logits,

hidden\_states=outputs.hidden\_states,

attentions=outputs.attentions,

)

<코드 끝>

[코드11]에 정의한 모델을 사용해서 질의응답 모델을 학습할 수 있다. 질의응답 모델의 학습은 research/squad/squad-training.ipynb를 참고하면 된다. 학습을 진행하면 아래와 같이 Loss가 감소하게 된다.

<블록 시작>

Average Loss = 1.1870): 100%|██████████| 8247/8247 [1:24:25<00:00, 1.63it/s]

Average Loss = 0.6109): 100%|██████████| 8247/8247 [1:24:30<00:00, 1.63it/s]

Average Loss = 0.2965): 100%|██████████| 8247/8247 [1:24:34<00:00, 1.63it/s]

Average Loss = 0.1559): 100%|██████████| 8247/8247 [1:24:33<00:00, 1.63it/s]

Average Loss = 0.1029): 100%|██████████| 8247/8247 [1:24:34<00:00, 1.63it/s]

Average Loss = 0.0817): 100%|██████████| 8247/8247 [1:24:31<00:00, 1.63it/s]

Average Loss = 0.0699): 100%|██████████| 8247/8247 [1:24:34<00:00, 1.63it/s]

Average Loss = 0.0608): 100%|██████████| 8247/8247 [1:24:37<00:00, 1.62it/s]

Average Loss = 0.0549): 100%|██████████| 8247/8247 [1:24:31<00:00, 1.63it/s]

Average Loss = 0.0497): 100%|██████████| 8247/8247 [1:24:43<00:00, 1.62it/s]

<블록 끝>

squad-training.ipynb에서 학습한 질의응답 모델을 squad\_model.bin에 저장하고, 이 파일을 이용해서 dev-v2.0.json 파일에 대한 추론을 하면 F1 스코어 73.30721044065311를 얻을 수 있다. squad\_model.bin을 활용해서 F1 스코어를 구하는 과정은 파이선 스크립트로 작성했다. [코드12]를 참고하면 된다.

<코드 시작>

코드12

$ python run\_evaluate.py --cache\_dir=caches --version\_2\_with\_negative

$ python evaluate.py data/dev-v2.0.json outputs/predictions\_.json

{

"exact": 68.97161627221426,

"f1": 73.30721044065311,

"total": 11873,

"HasAns\_exact": 66.24493927125506,

"HasAns\_f1": 74.92856099221858,

"HasAns\_total": 5928,

"NoAns\_exact": 71.69049621530698,

"NoAns\_f1": 71.69049621530698,

"NoAns\_total": 5945

}

<코드 끝>

SQuAD v2.0의 리더보드를 확인해보면 73.30은 약 77~78등에 해당하는 점수이다. 2018년 이후 BERT를 능가하는 다른 모델들이 많이 나온 결과로 현재 BERT의 랭킹은 높지 않지만 데이터셋을 준비하고 질의응답 모델을 디자인하는 노력이 적었다는 점에서 노력대비 성능이 좋은 모델을 만들 수 있었다. BERT 모델의 사이즈를 늘리거나 전처리하는 방법 또는 모델 구조를 조금 더 좋은 방법으로 변경할 경우 성능이 더 좋아질 수 있겠지만 이 절에서는 적은 노력 대비 좋은 성능의 모델을 만들 수 있다는 점에만 집중하려고 한다.

이번 장에서는 BERT의 구조와 BERT를 사전학습/파인튜닝 학습 방법을 자세히 알아봤다. BERT는 트랜스포머의 인코더 구조를 기본으로 하고 있으며, MLM과 NSP 기법을 이용해서 사전학습된다. 그리고 사전학습된 BERT를 로딩해서 여러가지 테스크(븐류, 질의응답 등등)로 파인튜닝할 수 있다는 것을 배웠다. 다음 절에서는 트랜스포머의 디코더를 활용하는 GPT에 대해서 알아보려고 한다.

4.3. GPT

앞 절에서 공부한 BERT는 Transformer의 인코더 부분을 이용해서 만든 모델이다. GPT는 Transformer의 디코더 부분을 이용해서 만든 모델이며 OpenAI가 2018년 6월 11일에 발표했다. BERT보다 약 4달 일찍 발표된 모델이다. GPT는 Generative Pre-Training의 약자이다. 우리말로 번역하면 생성하는 사전학습이다. GPT에서는 앞에 나온 단어를 이용해서 다음 단어를 맞춰나가는 방식으로 사전학습을 진행한다.

4.3.1. GPT의 사전학습

GPT는 주어진 단어를 기반으로 다음에 올 단어를 맞추는 방식으로 사전학습을 진행한다. 전통적인 언어 모델의 정의이다. GPT를 사전학습할 때 사용되는 입력 문장은 여러 개의 토큰으로 분리된다. 가령, "나는 오늘 학교에 갔다"라는 문장이 "나는", "오늘", "학교에", "갔다"와 같이 4개의 토큰으로 분리가 될 수 있다. GPT는 이렇게 분리된 토큰을 이용해서 언어 모델 학습을 한다.

<그림 시작>

그림13

<그림 끝>

[그림13]을 보면 4개의 토큰을 이용해서 언어모델 학습을 3번 진행했다는 것을 알 수 있다. 주어진 토큰들로부터 다음 토큰을 예측할 때 어텐션이 적용된다. 이 때 반드시 주어진 토큰들로부터만 어텐션 가중치를 가져야한다. [그림13]에서 "학교에"를 예측하는 경우를 생각해면, "학교에"를 예측하기 위해서 "나는"과 "오늘" 토큰으로부터 가중치를 각각 0.3, 0.7 정도 받아서 예측하게 된다. "학교에"를 예측할 때 "갔다" 토큰으로부터는 어떠한 정보도 받아오면 안된다. 그렇게 하기 위해서는 "갔다" 토큰에 대해서는 가중치를 0으로 만들어주면 된다. 주어진 토큰이 아닌 미래의 토큰으로부터 문맥 정보를 습득하지 못하게 하도록 가중치를 0으로 만들어주는 self-attention 기법을 masked self-attention이라고 한다.

GPT에서 masked self-attention을 사용하므로서 BERT와 큰 차이점을 가지게 된다. BERT의 경우, 한 위치에 해당하는 토큰을 예측하는 MLM을 실행할 때 필요한 정보를 앞/뒤 토큰들로부터 가져왔다. 양쪽 방향으로부터 정보를 가져올 수 있는 구조이다. BERT의 B가 양방향을 뜻하는 Bidirectional인 이유이다. 반면에 GPT는 예측할 토큰의 뒤에 나오는 토큰으로부터는 문맥 정보를 가져올 수 없다. 따라서 단방향(Unidirectional)한 성격을 가지고 있다.

4.3.2. masked self-attention

GPT가 masked self-attention을 통해서 예측할 토큰의 뒷쪽에 있는 토큰으로부터는 정보를 습득하지 못하게 한다는 것을 배웠다. 이것을 어떻게 구현할 수 있을까? 이번 절에서는 masked self-attention을 어떻게 구현하는지 알아보자. 우선 쿼리, 키, 값으로 어텐션 가중치를 구하는 과정을 다시 한번 그림으로 이해해보자. [그림11]에서 어텐션 가중치의 shape이 (B, N, M, M)이다. 여기에 매우 큰 음수(-10000)를 더해서 masked self-attention을 구현한다. [그림14]을 보자.

<그림 시작>

그림14

<그림 끝>

[그림14]에서 softmax 연산 후 결과를 보면 매우 큰 음수를 더한 영역의 값이 0으로 바뀌었다. 매우 큰 음수를 softmax 취했을 때 그 값은 0으로 수렴하게 된다. -10000을 더해준 이유가 바로 softmax 취했을 때 어텐션의 값을 0으로 수렴하게 인위적으로 만들어주기 위함이다. softmax를 취한 후에는 마스킹이 되지 않은 부분의 어텐션 가중치 합이 1이 된다. 마스킹한 후의 어텐션 가중치를 [그림5]에서와 같이 적용할 경우 앞에 나타난 토큰만을 이용해서 어텐션을 계산할 수 있게 된다.

4.4. RoBERTa

RoBERTa는 Facebook에서 2019년 7월 26일에 발표한 논문이다. 논문 제목은 "RoBERTa: A Robustly Optimized BERT Pretraining Approach"이다. 이 논문의 핵심은 BERT를 최적으로 학습하는 것이다. 논문에서는 BERT의 성능을 추가적으로 더 향상시킬 수 있는 방법을 소개했다. RoBERTa의 모델 구조는 BERT의 구조와 동일하지만 MLM을 학습할 때 데이터를 처리하는 방법, NSP를 처리하는 방법, 배치 사이즈 조절, 토크나이저 변경 등의 방법을 통해 몇몇 벤치마크 데이터셋 테스크에서 BERT의 성능을 뛰어넘었다.

4.4.1. Static/Dynamic 마스킹 전략

RoBERTa에서 MLM을 학습할 때 사용하는 마스킹 전략을 수정했다. Static 또는 Dynamic 두 가지 방법으로 수정했다.

원래의 BERT에서는 데이터 전처리 과정에서 토큰들에 대한 마스킹을 수행한다. 즉 하나의 데이터에 대해서 하나의 마스킹만이 사용된다. Static 마스킹 전략은 하나의 데이터를 10번 복재해서 각각 마스킹을 독립적으로 적용하는 것이다. 결과적으로 하나의 데이터에 마스킹을 서로 다르게 해줌으로서 서로 다른 10개의 데이터를 사용해서 MLM을 학습하게 된다.

Static 마스킹 전략보다 더 자유롭게 마스킹하는 방법이 Dynamic 마스킹이다. 모델에 입력으로 사용할 때마다 마스킹을 새로 하는 것이 Dynamic 마스킹이다.

[그림15]는 Static/Dynamic 마스킹의 예시를 보여준다.

<그림 시작>

그림15

서로 다르게 마스킹된 그림 static

서로 다르게 마스킹된 그림 dynamic

<그림 끝>

4.4.2. NSP 전략

RoBERTa에서는 BERT의 사전학습에서 사용하고 있는 NSP를 여러 가지 방법으로 변형시켜서 테스트했다.

<그림 시작>

그림16

SEGMENT-PAIR+NSP: 세그먼트 단위로 NSP 학습. 단, 세그먼트는 여러 자연어 문장으로 구성돼 있음

SENTENCE-PAIR+NSP: 자연어 문장으로 NSP 학습.

FULL-SENTENCES: 하나 이상의 문단에서 샘플링된 문장들로 512개의 토큰을 채운다.

DOC-SENTENCES: 하나의 문단에서 샘플링된 문장들로 512개의 토큰을 채운다.

<그림 끝>

[그림16]의 각 경우를 알아보자. SEGMENT-PAIR+NSP는 세그먼트 단위로 NSP를 진행한다는 것이다. SEGMENT는 하나의 데이터에서 연속된 문장이다. SENTENCE-PAIR+NSP는 문장 단위로 NSP를 하는 것이다. FULL-SENTECES와 DOC-SENTENCES에서는 NSP를 하지 않는다. FULL-SENTENCES의 경우 임의의 SEGMENT를 한개 이상의 문단으로부터 가져와서 길이가 512가 될 때까지 잇는것이다. DOC-SENTENCES는 FULL-SENTENCES와 같은 방법인데 한개의 문단으로부터만 데이터를 가져오는 방법이다.

4.4.3. 배치 사이즈와 데이터셋 크기

[그림16]에서 SENTENCE-PAIR+NSP와 DOC-SENTENCES의 경우에는 토큰의 길이가 보통 512보다 작을 확률이 크다. SENTECE-PAIR+NSP의 경우에는 문장 두 개만을 잇는 것이기 때문에 그렇다. DOC-SENTECES의 경우에는 하나의 문단에서 샘플링해서 샘플링한 지점의 이후 데이터가 모두 데이터가 되는 방법인데, 만일 샘플링된 지점이 문단의 거의 끝 부분이라면 토큰의 수가 512보다 훨씬 적아진다. 이런 경우 배치 사이즈를 늘려서 하나의 배치 사이즈에서 충분한 개수의 토큰 수가 나올 수 있도록 수정한 것이다. 또한 학습에 사용하는 데이터셋의 크기도 키웠다. BERT에서 사전학습에 사용한 데이터셋은 Wikipedia 데이터셋으로 약 16GB이다. RoBERTa에서는 그 외 다른 데이터셋을 추가해서 총 160GB로 늘렸다.

4.5. ALBERT

ALBERT는 2019년 7월에 Google Research와 Toyota Technological Institute at Chicago가 연구해서 발표한 언어 모델이다. ALBERT는 BERT의 모델 사이즈가 크다는 단점을 극복한 언어 모델이며, 모델의 사이즈를 크게 줄이면서 동시에 성능도 비슷하거나 높은 수준으로 올렸다. 모델의 사이즈를 줄이는 방법으로 두 가지 기법을 사용했는데, 이 절에서 그 기법에 대해서 자세하게 살펴볼 것이고 ALBERT를 사용했을 때 모델의 크기가 얼마나 줄어들었는지 알아보자.

4.5.1. Factorized embedding parameterization

ALBERT 논문에서 Factorized embedding parameterization은 임베딩 파라미터를 통해 모델의 크기를 줄이는 방법으로 소개하고 있다. 이 기법을 이해하기 위해서 우선 BERT에서 입력 데이터를 어떻게 임베딩하는지 다시 한번 되세겨보자. Factorized embedding parameterization을 이해하기 위해서는 5.2.1절을 선행으로 이해해야한다. 4.2.1절을 공부하지 않은 독자들은 5.2.1절을 먼저 공부하고 이 절을 다시 공부하길 바란다.

ALBERT에서는 임베딩 할 때 사용되는 파라미터의 개수를 조정하므로서 임베딩에 들어가는 파라미터 수를 줄였다. BERT에서는 임베딩 레이어를 만들때 임베딩 사이즈를(E) 히든 사이즈(H)와 같게 뒀다. H가 커질수록 E도 같이 커지게 되는 구조이다. H는 Context-dependent한 피처를 학습하기 위함인데 반해 E는 Context-independent한 피처를 학습하기 위한 피처이다. BERT는 context-dependent한 피처를 통해서 강력한 성능을 내는 모델이다. 따라서 H를 크게 갖는 것은 좋다. 하지만 그로 인해 E가 필요 이상으로 커지게 된다. 즉 BERT의 사전학습 과정에서 E는 부분적으로만 학습될 가능성이 있다는 것이다. ALBERT 논문에서는 E와 H를 굳이 같은 값으로 묶어버릴 필요가 없다고 이야기하고 있다.

<인용 시작>

https://arxiv.org/pdf/1909.11942.pdf

As such, untying the WordPiece embedding size E from the hidden layer size H

allows us to make a more efficient usage of the total model parameters as informed by modeling

needs

--> 워드피스 임베딩 사이즈 E를 히든 레이어 사이즈 H와 다른 값을 쓰면(untying) 전체 모델 파라미터를 더 효과적으로 사용할 수 있다.

<인용 끝>

BERT의 임베딩 레이어의 파라미터 크기는 VxE이다. 그런데 BERT에서는 E=H이기 떄문에 임베딩 레이어의 파라미터 크기는 VxH와 같다. ALBERT에서는 VxE로 임베딩하고 이렇게 임베딩된 E 차원을 H 사이즈로 다시 변형한다. 이때 E는 H보다 훨씬 작게 설정한다. 논문에서는 이 내용을 수학적으로 설명하고 있다. VxH를 분해(decomposition)해서 VxE와 ExH로 만든다고 설명한다. ALBERT와 BERT의 임베딩 사이즈를 비교해보면 아래와 가탇.

<블록 시작>

블록1

# BERT

V=30000

H=768

BERT의 임베딩 레이어 파라미터 개수 = 30,000 x 768 = 23,040,000

# ALBERT

V=30000

E=128

H=4096

ALBERT의 임베딩 레이어 파라미터 개수 = 30000 x 128 + 128 x 4096 = 3,840,000 + 524,288 = 4,364,288

<블록 끝>

[블록1]을 보면 ALBERT의 임베딩 레이어 파라미터 개수는 BERT에 비해 약 80% 줄었다. [코드13]을 통해 Transformers에서 구현된 ALBERT와 BERT에서 각각의 임베딩 사이즈를 확인해보자.

<코드 시작>

코드13

from transformers import BertModel, AlbertModel

bert = BertModel.from\_pretrained('bert-base-uncased')

albert = AlbertModel.from\_pretrained('albert-base-v2')

def num\_model\_param(m):

return sum(mi.numel() for mi in m.parameters())

albert\_embedding = num\_model\_param(albert.encoder.embedding\_hidden\_mapping\_in) + num\_model\_param(albert.embeddings)

bert\_embedding = num\_model\_param(bert.embeddings)

print('number of BERT Embedding parameters: {}'.format(bert\_embedding))

print('number of ALBERT Embedding parameters: {}'.format(albert\_embedding))

<코드 끝>

<블록 시작>

블록2

number of BERT Embedding parameters: 23837184

number of ALBERT Embedding parameters: 4005120

<블록 끝>

[블록2]를 보면 BERT의 임베딩 파라미터 개수는 ALBERT의 임베딩 파라미터 개수의 약 16.8%이다.

4.5.2. Cross-layer parameter sharing

BERT의 파라미터 개수를 비약적으로 줄인 것은 Cross-layer parameter sharing이다. BERT에서 Self-Attention을 계산해서 H차원의 결과값을 만들어내는 BertLayer 블록을 12번 반복한다. 이때 파라미터를 공유하지 않기 때문에 같은 구조의 블록을 12개 만든다. ALBERT에서는 이 구조의 블록을 한번만 만들되, 사용시에는 결과값을 다시 입력값으로 집어넣는 과정을 12번 반복해서 결과적으로 모델의 사이즈를 1/12로 줄인다. 이 과정에서 처음에 만든 구조를 반복해서 사용하기 때문에 파라미터를 공유하게 되는 것이다. [코드14]은 BERT와 ALBERT에서 인코더 동작 구조를 의사 코드로 작성한 것이다.

<코드 시작>

코드14

# BERT

class BertLayer:

# BERT에서 Self-Attention을 계산해서 H차원의 결과값을 만들어내는 레이어

...

# ALBERT

class AlbertLayer:

# ALBERT에서 Self-Attention을 계산해서 H차원의 결과값을 만들어내는 레이어

...

bert\_layers = [BertLayer() for \_ in range(num\_layers)

albert\_layer = AlbertLayer()

for i in range(num\_layers):

x = bert\_layers[i](x)

for i in range(num\_layers):

x = albert\_layer(x)

<코드 끝>

[코드14] 의사코드를 보면 BERT에서의 bert\_layers는 [i]를 통해 인덱싱해서 num\_layers개의 서로 다른 BertLayer를 사용하게 된다. 반면에 ALBERT에서는 AlbertLayer 한 개를 만들어서 그것을 반복해서 사용하고 있다. Transformers에서 구현된 ALBERT와 BERT에서 각각의 인코더의 사이즈를 확인해보자.

<코드 시작>

코드15

from transformers import BertModel, AlbertModel

bert = BertModel.from\_pretrained('bert-base-uncased')

albert = AlbertModel.from\_pretrained('albert-base-v2')

# 모델의 파라미터를 구하는 함수

def num\_model\_param(m):

return sum(mi.numel() for mi in m.parameters())

bert\_encoder = num\_model\_param(bert.encoder)

albert\_encoder = num\_model\_param(albert.encoder)

print('number of BERT Encoder parameters: {}'.format(bert\_encoder))

print('number of ALBERT Encoder parameters: {}'.format(albert\_encoder))

<코드 끝>

[코드15]를 실행하면 [블록3]의 결과를 확인할 수 있다.

<블록 시작>

블록3

number of BERT Encoder parameters: 85054464

number of ALBERT Encoder parameters: 7186944

<블록 끝>

BERT 인코더 파라미터의 개수는 ALBERT 인코더 파라미터 개수의 약 11.8배이다. 임베딩과 인코더의 파라미터 개수를 비교하는 코드는 research/chapter5/parameter\_count\_comparision.ipynb을 통해 확인하면 된다.

여기에서 한번 비판적으로 생각해보자. Cross-layer parameter sharing 기법에서 파라미터 개수를 줄이는 원리는 굉장히 간단하다. 하지만 ALBERT처럼 하나의 AlbertLayer를 만들어서 그것을 반복해서 사용해도 모델의 성능상 문제가 없을까? [코드14]에서 AlbertLayer 한 개로 입력 x값을 계속 업데이트하고 있다. for 문의 각 단계에서 x가 업데이트되기 전과 후의 L2 거리와 코사인 거리를 구해서 그것을 그래프로 그리면 [그림17]과 같이 그릴 수 있다.

<그림 시작>

그림17

ALBERT paper Figure 1

<그림 끝>

[그림17]은 입력 값 x를 AlbertLayer에 넣어서 계속 업데이트시키면 입력 x와 업데이트된 x의 차이가 점차 줄어들게 된다는 것을 보여주는 그림이다. ALBERT 논문에서는 이렇게 입력과 출력의 차이가 줄어드는 지점에 도달하는 것을 평행점(equilibrium point)에 도달한다고 표현하고 있다. 모델의 파라미터를 공유하므로서 평행점에 도달하면 같은 구조의 모델도 더 높은 성능을 보인다고 설명하고 있다.

<인용 시작>

https://arxiv.org/pdf/1909.11942.pdf

Similar strategies have been explored by Dehghani et al. (2018) (Universal Transformer, UT) and

Bai et al. (2019) (Deep Equilibrium Models, DQE) for Transformer networks. Different from our

observations, Dehghani et al. (2018) show that UT outperforms a vanilla Transformer.

<인용 끝>

research/chapter5/albert\_parameter\_sharing.ipynb를 보면 [그림17]과 같이 입력 x와 업데이트된 x의 차이가 점차 줄어들게 되는 것을 보여주고 있다.

4.5.3. Sentence order prediction

ALBERT에서 또 하나 집고 넘어가야 할 부분이 SOP(Sentence Order Prediction)이다. BERT에서의 NSP가 언어 모델을 학습하는데 있어서 유용한지에 대한 의구심이 있어서 NSP를 조금 다른 방식으로 변형한 것이다. SOP는 두 개의 연결된 문장을 사용한다. 그리고 문장의 순서를 바꾸지 않으면 1, 바꾸면 0으로 정의하고 학습한다.

<블록 시작>

블록4

예제 문단: 아침에 일어났다. 이상하게 배가 아팠다. 그래서 병원에 갔다.

positive example: <CLS> 이상하게 배가 아팠다. <SEP> 그래서 병원에 갔다. <SEP>

negative example: <CLS> 그래서 병원에 갔다. <SEP> 이상하게 배가 아팠다. <SEP>

<블록 끝>

ALBERT를 사전학습할 때 SOP를 학습하기 위해서 [블록4]와 같이 <CLS>와 <SEP>을 문장 앞뒤와 중간에 붙여준다. BERT의 NSP도 이와 같이 학습된다.

4.5.4. ALBERT 정리

ALBERT는 BERT 모델의 성능을 비슷하거나 높은 수준으로 향상시켰고, 동시에 모델의 사이즈는 약 1/10으로 줄였다. 모델의 사이즈를 줄이는 방법은 두 가지인데, 하나는 Factorized embedding parameters이고 다른 하나는 Cross-Layer parameter sharing이다. 또한 사전학습을 진행할 때도 NSP를 SOP로 변형했다.

<표 시작>

chapter5.xlsx

t1 sheet

<표 끝>

4.6. ELECTRA

ELECTRA(Efficiently Learning an Encoder that Classifies Token Replacements Accurately)는 Google Brain과 스탠포드 대학교가 함께 연구해서 2020년 5월 23일에 발표한 언어 모델이다. 일반적으로 언어 모델을 사전학습할때 학습 시간이 오래 걸린다. ELECTRA는 학습의 효율을 개선함으로서 사전학습 시간을 현저하게 줄였다. RoBERT, ALBERT, BERT에서 언어 모델을 학습할 때 MLM을 사용했는데, ELECTRA는 RTD(Replaced Token Detection)이라는 방법을 사용한다. 이번 절에서 ELECTRA에 대해서 자세하게 이해해보자.

4.6.1. 학습구조

ELECTRA는 GAN(Generative Adversarial Networks)과 유사한 구조를 가지고 있다. GAN에 대해서 간략하게 설명을 하고 넘어가자. GAN은 Generator와 Discriminator로 구성된다. 입력 값을 Generator가 생성(가짜 입력값)하고 Discriminator는 Generator가 준 입력값이 진짜(1)인지 가짜(0)인지 구별하는 학습을 한다. 이렇게 계속학습을 하다보면 Discriminator가 진짜와 가짜를 구별하지 못하게 되고 Generator는 정말 진짜 같은 데이터를 만들어내는 네트워크로 학습된다.

ELECTRA를 학습할 때도 Generator와 Discriminator가 필요하다. ELECTRA 논문에서 사용하고 있는 그림을 보자.

<그림 시작>

그림18

https://arxiv.org/pdf/2003.10555.pdf

Figure2

<그림 끝>

입력 값 "the chef cooked the meal"을 generator가 "the chef ate the meal"로 바꿨다. 이 때 MLM을 이용해서 바꾼다. Discriminator는 어떤 단어가 바뀐 단어인지 알아내는 학습을 한다(RTD). 학습이 완료된 Discriminator가 ELECTRA 모델이다. 이 때 Generator는 사용하지 않는다. Generator를 직접 사용하지 않기 때문에 Generator의 크기가 그렇게 클 필요가 없다.([그림18]의 Generator에 "typically a small MLM")

4.6.2. RTD

ELECTRA는 학습의 효율을 개선함으로서 사전학습 시간을 현저하게 줄였다. 학습의 효율을 개선한 비밀이 RTD이다. MLM은 전체 입력 토큰의 15%만 랜덤으로 정해서 그 부분에 대해서만 학습을 진행한다. 나머지 85%에 대해서는 학습이 진행되지 않는다. ELECTRA의 저자들은 이 부분이 비효율적이라고 생각한 것이다. 그래서 각 토큰에 대해서 학습을 진행할 수 있는 방법인 RTD를 소개한 것이다. 5.5.1의 그림7에서 알 수 있듯이, Discriminator에서 RTD를 학습하는데 각 토큰에 대해서 그 토큰이 fake인지 real인지 판단한다. 즉 모든 토큰들이 학습 대상이 된다. 하나의 입력 데이터로 더 많은 학습을 할 수 있게 된다.

<그림 시작>

그림19

https://arxiv.org/pdf/2003.10555.pdf

Figure1

<그림 끝>

[그림19]을 보면 여러 언어 모델들과 ELECTRA 기반의 모델들이 특정 GLUE 스코어에 도달할 때까지의 FLOPs를 그래프로 보여주고 있다. FLOPs는 Floating Point Operations의 약자이다. [그림19]의 단위를 보면 1e-20이다. 무려 1천 경에 해당하는 숫자이다. 가령, BERT-large의 경우 약 2e-20번의 Floating Point 연산을 했다는 것이다. ELECTRA-large와 비교했을 때 거의 같은 FLOPs이지만 도달한 GLUE 스코어는 매우 차이가 많이 난다.

4장에서 BERT의 구조와 학습 방법에 대해서 자세하게 알아봤다. 또한 BERT 이후에는 어떤 모델이 나왔는지(RoBERTa, ALBERT, ELECTRA)에 대해서 알아봤다. 5장에서 알아본 모든 모델들이 기본적으로 Transformer를 기반으로 하고 있다. 2018년 이후의 모든 언어 모델은 기본적으로 Transformer 구조를 내부적으로 가지고 있다. Transformer를 기반으로 RoBERTa는 사이즈를 키우고 불필요한 학습을(NSP) 없애서 성능을 높였고, ALBERT는 모델의 크기를 줄였고, ELECTRA는 모델의 학습 시간을 단축했다. 같은 기반 기술을 가지고 관점을 다르게 해서 서로 다른 언어 모델을 사전학습한 것이다. NLP를 연구할 때 사전학습된 모델을 로딩한 후 파인튜닝해서 Task-specific한 모델을 만드는 트랜드로 변한 것이 BERT 이후에 가장 큰 변화이다.

### Reference

- 유투브 비디오: https://www.youtube.com/watch?v=q9NS5WpfkrU

- MLM 실제 전략: https://towardsml.com/2019/09/17/bert-explained-a-complete-guide-with-theory-and-tutorial/