fewshot/oneshot/zeroshot 개념/코드

GPT2 데이터셋, FS/OS/ZS 개념 -> not very detail, show an example play:plays|talk:talks|say:?

------------------------------

6. 파인튜닝 없는 언어모델: GPT2 그리고 GPT3

6.1. 학습을 위한 학습, 메타 러닝

6.2. 메타 러닝을 이용한 Amazon 리뷰 감정 분류 학습하기

6.3. GPT2 그리고 GPT3

<블록 시작>

이제 말을 막 배워가는 윤우에게 엄마 아빠는 조금씩 어려운 말을 알려주고 싶어졌다.

엄마: 윤우야, 엄마 이름은 XXX야, 아빠 이름은 OOO야, 윤우 이름은 뭐야?

윤우: 윤우에여

아빠: 윤우야, 아빠는 xx살이고, 엄마는 xx살이야, 윤우는 몇살이야?

윤우: 세 살이에여

<블록 끝>

2018년부터 2020년까지의 거의 모든 언어 모델은 Transformer를 기반의 모델이었다. 이 언어 모델들은 사전학습돼서 사전학습된 모델의 형태로 존재한다. 사전학습된 모델은 파인튜닝 학습을 통해 다양한 형태의 자연어 처리 모델로 학습될 수 있다. 이 때 파인튜닝을 위한 학습데이터가 필요하다. 파인튜닝은 이미 학습된 상태에서 추가적으로 하는 학습이기 때문에 학습데이터가 상대적으로 적어도 된다는 것이 파인튜닝의 장점이다. 하지만 여전히 데이터셋을 준비하는 것은 성가신 일이다. 현업에서 자연어 처리 모델을 만들 때 현업의 특성에 맞는 잘 분포된 데이터를 얻기 힘들다.

2019년 2월에 OpenAI에서 GPT2를 발표했다. 논문의 제목은 "Language Models are Unsupervised Multitask Learners"이다. 이 논문에서 파인튜닝을 통한 접근법은 잘 일반화된 모델을 만들기 힘들다고 이야기하고 있다.

<인용 시작>

인용1

Our suspicion is that the prevalence of single task training

on single domain datasets is a major contributor to the lack

of generalization observed in current systems.

... (중략) ...

Multitask learning (Caruana, 1997) is a promising framework for improving general performance

https://d4mucfpksywv.cloudfront.net/better-language-models/language-models.pdf

<인용 끝>

[인용1]에서는 하나의 도메인으로 이루어진 데이터셋에서 하나의 테스크를 학습하기 때문에 일반화가 잘 안된다고 이야기하고 있고, 그것을 해결하기 위해서 다양한 테스크를(multitask) 학습할 필요가 있다고 한다. GPT2에서는 다양한 도메엔에서의 학습이 zero-shot 세팅을 통해 진행됐다. zero-shot 세팅에 대한 구체적인 구현 방법은 OpenAI에서 공개하지 않았기 때문에 이 장에서 다루지 않으려고 한다. 이번 장에서는 NLP에서의 메타 러닝(meta learning)을 공부해보고 간단하게 코드를 통해서도 알아보려고 한다.

6.1. 학습을 위한 학습, 메타 러닝

세상의 모든 데이터를 포함하고 있는 데이터가 있을까? 그런 데이터는 존재하지 않는다. 가령 컴퓨터 비전 분야의 연구에서 가장 많이 사용되고 있는 데이터셋으로 IMAGENET 데이터셋이 있다. 이 IMAGENET 데이터셋은 약 1400만개의 데이터셋을 1000개의 클래스로 분류하고 있는 데이터셋이다. 그렇다면 이 데이터셋이 전세계의 모든 이미지를 분류할 수 있을 만큼 다양할까? 그렇지 않다. 예를 들어보자. 사용자가 올리는 이미지를 AI 모델로 자동으로 분류하는 모델로 서비스를 하고 있는 중고거래 사이트가 있다고 해보자. 이 사이트는 2000년도부터 운영돼 왔는데 2010년이 되면서 스마트폰이라는 물건이 생기면서 사람들이 서로의 스마트폰을 중고거래 사이트를 통해서 거래하기 시작했다고 하자. 이 중고거래 사이트가 처음 운영되기 시작한 2000년도에 학습된 모델은 스마트폰이라는 물건을 한번도 학습한 적이 없었을 것이고, 따라서 스마트폰을 Phone이라는 카테고리로 분류할 수 없을 것이다. 그렇다면 스마트폰 이미지만 따로 모아서 스마트폰을 분류할 수 있는 모델을 새로 만들어서 새로 서비스하면 된다. 그런데 스마트폰 이미지를 모아서 일일이 레이블링하는 과정은 매우 지루한 일이다. 더욱이 스마트폰과 같이 기존에는 없었던 형태의 물건은 언제든지 나올 수 있는데, 그때마다 데이터셋을 새로 모아서 레이블링을 한다면 매우 비효율적일 것이다.

이 문제를 해결할 수 있는 방법이 바로 메타 러닝이다. 일반적인 기계학습에서의 학습과 메타 러닝을 이미지를 통해 간단하게 설명해보면 [그림1]과 같다.

<그림 시작>

그림1

중고거래 사이트 이미지 분류 모델 학습/평가 데이터셋 (기계학습 vs 메타러닝)

<그림 끝>

[그림1]에서 전통적인 기계학습의 학습과 평가 데이터셋을 보면 같은 클래스의 데이터로 구성돼 있다. 학습에서도 전화기/컵 데이터셋을 사용하고, 평가에서도 전화기/컵 데이터셋을 사용한다. 메타 러닝에서는 학습할 때마다 테스트할 때마다 데이터셋을 구성하는 클래스가 다르다. 메타 러닝을 학습할 때 전화기/컵 데이터셋과 공/막대기 데이터셋과 같이 서로 다른 구성의 학습 데이터셋을 사용한다. 테스트할 때 역시 쥐/늑대와 같이 전혀 다른 클래스의 데이터를 이용해서 학습한다. 메타 러닝은 학습하는 법을 학습하는 것이라고 소개된다. [그림1]에서와 같이 전화기/컵, 공/막대기와 같이 서로 다른 구성의 학습 데이터를 이용해서 구분할 줄 알도록 모델을 학습할 경우, 전혀 새로운 데이터셋인 자전거/의자와 같은 데이터가 입력으로 들어와도 그 둘을 구분해낼 수 있게 된다.

이 절에서 사용한 중고거래 사이트 예시로 돌아가보자. 모델을 처음 만들었을 때 메타 러닝을 고려해서 만들었다면 2010년도에 스마트폰이 나온 후에 추가적인 학습을 하지 않아도 된다. 간단하게 스마트폰 사진 몇 장과 스마트폰이 아닌 다른 클래스의 사진 몇 장을 모델에 입력으로 넣어주면 스마트폰인지 아닌지 구분할 수 있게 된다. [그림2]를 보자.

<그림 시작>

그림2

<그림 끝>

스마트폰에 대한 학습이 전혀 진행되지 않았어도 [그림2]와 같이 스마트폰을 분류할 수 있다. [그림2]의 모델은 메타 러닝 기법을 이용해서 학습된 모델이기 때문에 서로 다른 클래스로 구성된 이미지셋을 입력으로 넣었을 경우 같은 분류의 클래스끼리 묶어줄 수 있다. 서로 다른 클래스의 이미지를 구분할 수 있는 능력을 학습한 모델인 셈이다. 그래서 메타 러닝을 학습하는 방법을 학습하는 것이라고 정의하기도 한다.

메타 러닝은 쉽지 않다. 특히 자연어처리 영역에 있어서 메타 러닝은 GPT2나 GPT3를 제외하고는 크게 주목받은 연구는 없었다. 이번 절에서는 메타 러닝 개념을 우리의 생활에 친숙한 예제를 통해서 알아봤다. 다음 절에서는 코드를 통해서 메타 러닝을 공부해보자.

6.2. 메타 러닝을 이용한 Amazon 리뷰 감정 분류 학습하기

이번 절에서는 메타 러닝을 이용해서 Amazon 리뷰 데이터셋에 대한 감정 분류 모델을 학습해보려고 한다. Amazon 리뷰 데이터셋은 Amazon에서 판매하는 물품을 종류별로 구분한 뒤 각 종류별로 긍정을 나타내는 후기 또는 부정을 나타내는 후기가 모아져있다. 이 데이터셋은 22개의 물품 리스트에 대한 데이터를 가지고 있다.

<블록 시작>

apparel, automotive,baby,beauty,camera\_photo,cell\_phones\_service,computer\_video\_games,health\_personal\_care,

magazines,music,software,sports\_outdoors,toys\_games,video,books,dvd,electronics,kitchen\_housewares,grocery,

office\_products,outdoor\_living,gourmet\_food,jewelry\_watches

<블록 끝>

각 물품들에 대해서 \*.train, \*.dev, \*.test 형태로 저장돼 있다. 예를 들어서 tools\_hardware에 대한 데이터셋은 [블록1]과 같이 train, dev, test셋으로 나뉘어 있다. 그리고 각 데이터셋에는 리뷰 데이터가 자연어 형태로 있고 각 데이터는 1(positive)과 -1(negative)로 레이블링 돼 있다.

<블록 시작>

블록1

$ ls -al Amazon\_few\_shot/tools\_hardware.\*

-rw-rw-r-- 1 953 9월 15 05:19 Amazon\_few\_shot/tools\_hardware.t2.dev

-rw-rw-r-- 1 2159 9월 15 05:19 Amazon\_few\_shot/tools\_hardware.t2.test

-rw-rw-r-- 1 10680 9월 15 05:19 Amazon\_few\_shot/tools\_hardware.t2.train

-rw-rw-r-- 1 1673 9월 15 05:19 Amazon\_few\_shot/tools\_hardware.t4.dev

-rw-rw-r-- 1 2744 9월 15 05:19 Amazon\_few\_shot/tools\_hardware.t4.test

-rw-rw-r-- 1 7064 9월 15 05:19 Amazon\_few\_shot/tools\_hardware.t4.train

-rw-rw-r-- 1 978 9월 15 05:19 Amazon\_few\_shot/tools\_hardware.t5.dev

-rw-rw-r-- 1 1206 9월 15 05:19 Amazon\_few\_shot/tools\_hardware.t5.test

-rw-rw-r-- 1 10356 9월 15 05:19 Amazon\_few\_shot/tools\_hardware.t5.train

$ cat tools\_hardware.t5.dev

i bought this item idea saving time it worked great until i dropped it , which three hours into day , cheaply made star wheels score paper bent over now you imagine it leaves wavy line instead nice straight one , i am very disapointed quality this product would not buy -1

japan this best item examine pupil reflex light see palate or tonsils 1

this product so great , so price , its great bargain so usefull around house my cars -1

used supra locks which i like lot , i found master lock key safe tried it out . it very heavy duty , holds much more than supra . combo easy change ( takes seconds ) it very weather proof . i like it better than supra except you must leave combination dialed close it . someone could see it . supra not show which button(s ) pushed . all things considered , i buying master lock future . 1

bessey k-body clamps superior multi-board cabinet door glue-ups keeping pieces flat . wish i could purchased k-body clamps instead all pipe clamps i own 1

<블록 끝>

이 데이터셋을 이용해서 메타 러닝 모델을 학습해보자. 우선 이 데이터셋에서 다루고 있는 물품을 학습/검증/테스트용으로 나눌 것이다. 레포지토리의 chapter6/train.list에 있는 물품들에 대해서는 메타 러닝 모델을 학습할 때 사용할 것이고 chapter6/dev.list에 있는 물품 데이터셋들을 이용해서 학습 중에 검증을 할 것이다. 학습이 완료된 후에는 chapter6/test.list에 있는 물품 데이터셋을 이용해서 테스트를 할 것이다. 각 파일의 내용은 [블록3]을 참고하라.

<블록 시작>

블록3

$ cat train.list

apparel

automotive

baby

beauty

camera\_photo

cell\_phones\_service

computer\_video\_games

health\_personal\_care

magazines

music

software

sports\_outdoors

toys\_games

video

$ cat dev.list

grocery

office\_products

outdoor\_living

gourmet\_food

jewelry\_watches

$ cat test.list

books

dvd

electronics

kitchen\_housewares

<블록 끝>

메타 러닝을 학습하기 위해서 N-way, K-shot 방식을 사용하려고 한다. N-way의 N은 클래스의 개수이다. Amazon 리뷰 데이터셋에서는 positive/negative 두 개의 클래스를 구분하기 때문에 N은 2이다. K-shot의 K는 서포트 데이터셋의 개수이다. 6.1절에서 메타 러닝은 학습하는 것을 학습하는 기법이라고 소개했는데 이것을 가능하게 하는 방법 중 하나가 서포트 데이터셋을 활용하는 것이다. 메타 러닝에서 하나의 배치 데이터셋은 서포트 데이터셋과 쿼리 데이터셋으로 나뉘어 있다. 메타 러닝 모델은 쿼리 데이터와 서포트 데이터의 관계를 학습한다. [그림3]을 보면 하나의 배치 데이터셋이 64개로 이루어져 있다. 앞에 10개는 positive/negative 데이터셋을 각각 5개씩 가지고 있고, 뒤에 54개가 쿼리 데이터셋으로 positive/negative 데이터가 각각 27개씩 있다. 이번 절에서 사용하는 메타 러닝 모델은 각 배치에 대하여 서포트 데이터와 쿼리 데이터를 각각 인코딩한 후, 인코딩된 서포트 데이터셋과 인코딩된 쿼리 데이터를 이용해서 쿼리 데이터의 정답을 맞추는 방법으로 학습된다.

<그림 시작>

그림3

<그림 끝>

이렇게 학습된 메타 러닝 모델은 새로운 물품에 대한 리뷰 데이터셋에도 쓰일 수 있다. 다만 몇 개의 서포트 데이터가 필요하다. 메타 러닝으로 학습돼 있지 않다면 새로운 물품에 대한 학습/검증/테스트 데이터셋을 최소 몇 만건 정도 수집해야 하며 수집 후에도 모델을 새로 학습해서 시스템에 새로 적용해야 한다. 따라서 메타 러닝 방식이 훨씬 효율적이라고 할 수 있다.

이제는 실제로 Amazon 리뷰 데이터셋을 메타 러닝하는 모델을 구현해보자. chapter6/few-shot-classification.ipynb을 참고하면된다.

6.2.1. 데이터셋/데이터로더 만들기

우선 데이터셋과 데이터로더를 만들어보자.

<코드 시작>

코드1

class AmazonDataset():

def \_\_init\_\_(self, data\_path, tokenizer, dtype):

self.data\_path = data\_path

self.tokenizer = tokenizer

with open(f'{dtype}.list', 'r') as f:

self.categories = [oneline.rstrip() for oneline in f]

self.support\_dataset = {}

self.dataset = {}

for category in tqdm(self.categories, desc='reading categories'):

self.dataset[category] = {

'neg': self.get\_data(category, 'neg', dtype),

'pos': self.get\_data(category, 'pos', dtype)

}

if dtype == 'test' or dtype == 'dev':

for category in tqdm(self.categories, desc='reading categories for support'):

self.support\_dataset[category] = {

'neg': self.get\_data(category, 'neg', 'train'),

'pos': self.get\_data(category, 'pos', 'train'),

}

def read\_files(self, category, label, dtype):

data = {

'text': [],

'label': []

}

for t in ['t2', 't4', 't5']:

filename = f'{category}.{t}.{dtype}'

with open(os.path.join(self.data\_path, filename), 'r') as f:

for oneline in f:

oneline = oneline.rstrip()

text = oneline[:-2]

if int(oneline[-2:]) == 1 and label == 'pos':

tensor = self.tokenizer(text, return\_tensors='pt')

data['text'].append(tensor['input\_ids'][0])

data['label'].append(1)

elif int(oneline[-2:]) == -1 and label == 'neg':

tensor = self.tokenizer(text, return\_tensors='pt')

data['text'].append(tensor['input\_ids'][0])

data['label'].append(0)

data['label'] = torch.tensor(data['label'])

return data

def get\_data(self, category, label, dtype):

data = self.read\_files(category, label, dtype)

return data

<코드 끝>

[코드1]은 Amazon 리뷰 데이터셋을 읽어서 AmazonDataset 객체로 만드는 클래스이다. AmazonDataset을 학습/검증/테스트 용으로 용도를 나눠서 각각 따로 객체를 만들었다. [코드2]를 참고하라.

<코드 시작>

코드2

train\_dataset = AmazonDataset(data\_path, tokenizer, 'train')

dev\_dataset = AmazonDataset(data\_path, tokenizer, 'dev')

test\_dataset = AmazonDataset(data\_path, tokenizer, 'test')

<코드 끝>

각각의 AmazonDataset 객체를 이용해서 데이터로더를 만들어보자. [코드3]은 AmazonDataLoader 클래스를 정의하는 \_\_init\_\_ 함수에 대한 구현이다.

<코드 시작>

코드3

class AmazonDataLoader():

def \_\_init\_\_(self, dataset, batch\_size, n\_support):

assert n\_support % 2 == 0, 'n\_support should be multiple of 2'

self.dataset = dataset

self.batch\_size = batch\_size

self.n\_support = n\_support

self.neg\_idx = {k:0 for k in dataset.dataset}

self.pos\_idx = {k:0 for k in dataset.dataset}

self.neg\_len = {k:len(dataset.dataset[k]['neg']['text']) for k in dataset.dataset}

self.pos\_len = {k:len(dataset.dataset[k]['pos']['text']) for k in dataset.dataset}

self.neg = {k:dataset.dataset[k]['neg'] for k in dataset.dataset}

self.pos = {k:dataset.dataset[k]['pos'] for k in dataset.dataset}

self.idx = 0

self.categories = [k for k in dataset.dataset]

# prepare for test dataset, support dataset should come from "\*.train"

self.neg\_support\_idx = {}

self.pos\_support\_idx = {}

self.neg\_support\_len = {}

self.pos\_support\_len = {}

if self.dataset.support\_dataset:

self.neg\_support\_idx = {k:0 for k in self.dataset.support\_dataset}

self.pos\_support\_idx = {k:0 for k in self.dataset.support\_dataset}

self.neg\_support\_len = {k:len(self.dataset.support\_dataset[k]['neg']['text']) for k in self.dataset.support\_dataset}

self.pos\_support\_len = {k:len(self.dataset.support\_dataset[k]['pos']['text']) for k in self.dataset.support\_dataset}

<코드 끝>

메타 러닝의 학습이나 테스트 과정에서 데이터를 배치 단위로 만드는 코드는 [코드4]를 참고하라. [코드4]는 학습을 위한 배치를 생성하는 코드이다.

<코드 시작>

코드4

def get\_batch(self):

category = self.categories[self.idx % len(self.categories)]

neg = self.neg[category]

pos = self.pos[category]

neg\_start\_idx = self.neg\_idx[category] % self.neg\_len[category]

pos\_start\_idx = self.pos\_idx[category] % self.pos\_len[category]

# prepare negative/positive dataset

neg\_text = neg['text'][neg\_start\_idx:neg\_start\_idx+(self.batch\_size//2)]

pos\_text = pos['text'][pos\_start\_idx:pos\_start\_idx+(self.batch\_size//2)]

neg\_label = neg['label'][neg\_start\_idx:neg\_start\_idx+(self.batch\_size//2)]

pos\_label = pos['label'][pos\_start\_idx:pos\_start\_idx+(self.batch\_size//2)]

self.neg\_idx[category] += (self.batch\_size//2)

self.pos\_idx[category] += (self.batch\_size//2)

if len(neg\_text) + len(pos\_text) != self.batch\_size:

return self.get\_batch()

# padding text dataset

neg\_text = pad\_sequence([n for n in neg\_text], batch\_first=True)

pos\_text = pad\_sequence([p for p in pos\_text], batch\_first=True)

neg\_text, pos\_text = pad\_text(neg\_text, pos\_text)

# prepare support/query text

neg\_support\_text = neg\_text[:self.n\_support//2]

pos\_support\_text = pos\_text[:self.n\_support//2]

neg\_query\_text = neg\_text[self.n\_support//2:]

pos\_query\_text = pos\_text[self.n\_support//2:]

# prepare support/query label

neg\_support\_label = neg\_label[:self.n\_support//2]

pos\_support\_label = pos\_label[:self.n\_support//2]

neg\_query\_label = neg\_label[self.n\_support//2:]

pos\_query\_label = pos\_label[self.n\_support//2:]

# merge support/query text

support\_text = torch.cat([neg\_support\_text, pos\_support\_text], dim=0)

query\_text = torch.cat([neg\_query\_text, pos\_query\_text], dim=0)

# merge support/query label

support\_label = torch.cat([neg\_support\_label, pos\_support\_label], dim=0)

query\_label = torch.cat([neg\_query\_label, pos\_query\_label], dim=0)

# make data and label

data = torch.cat([support\_text, query\_text], dim=0)

label = torch.cat([support\_label, query\_label], dim=0)

# increase category index

self.idx += 1

return data, label

<코드 끝>

AmazonDataLoader를 이용해서 배치 데이터를 생성해내는 코드는 [코드5]를 참고하자.

<코드 시작>

코드5

train\_dataset = AmazonDataset(data\_path, tokenizer, 'train')

dev\_dataset = AmazonDataset(data\_path, tokenizer, 'dev')

test\_dataset = AmazonDataset(data\_path, tokenizer, 'test')

>>> for i in range(10):

>>> d, l = train\_dataloader.get\_batch()

>>> print(d.shape, l.float().mean())

torch.Size([64, 149]) tensor(0.5000)

torch.Size([64, 460]) tensor(0.5000)

torch.Size([64, 254]) tensor(0.5000)

torch.Size([64, 262]) tensor(0.5000)

torch.Size([64, 1283]) tensor(0.5000)

torch.Size([64, 1658]) tensor(0.5000)

torch.Size([64, 613]) tensor(0.5000)

torch.Size([64, 359]) tensor(0.5000)

torch.Size([64, 530]) tensor(0.5000)

torch.Size([64, 602]) tensor(0.5000)

<코드 끝>

Amazon 리뷰 데이터셋의 데이터셋과 데이터로더를 만들었다. 이제 메타 러닝을 위한 모델 클래스를 구현해보자.

<코드 시작>

코드6

class FewShotInduction(nn.Module):

def \_\_init\_\_(self, C, S, vocab\_size, embed\_size, hidden\_size, d\_a,

iterations, outsize, weights=None):

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

self.encoder = Encoder(C, S, vocab\_size, embed\_size, hidden\_size, d\_a, weights)

self.induction = Induction(C, S, 2 \* hidden\_size, iterations)

self.relation = Relation(C, 2 \* hidden\_size, outsize)

def forward(self, x):

support\_encoder, query\_encoder = self.encoder(x) # (k\*c, 2\*hidden\_size)

class\_vector = self.induction(support\_encoder)

probs = self.relation(class\_vector, query\_encoder)

return probs

<코드 끝>

[코드6]을 보면 Encoder가 배치로 입력되는 데이터 x를 인코딩해서 support\_encoder와 query\_encoder로 나눠준다. support\_encoder를 Induction 클래스를 이용해서 class\_vector를 생성하고, class\_vector와 query\_encoder를 이용해서 확률 probs를 생성한다. Encoder, Induction 그리고 Relation 서브 클래스의 구현은 chapter6/model.py를 참고하면 된다.

<코드 시작>

코드7

support = 5

model = FewShotInduction(C=2,

S=support,

vocab\_size=30522,

embed\_size=300,

hidden\_size=128,

d\_a=64,

iterations=3,

outsize=100)

model = model.cuda()

<코드 끝>

메타 러닝 모델은 [코드7]과 같이 정의할 수 있다. support 데이터로 사용할 개수를 5개로 정의해서 넣어주는데, 이 파라미터를 통해서 Encoder 클래스가 서포트 데이터는 몇 개 사용하는지를 알 수 있다.

학습을 시작하기 전에 [코드8]에서 Criterion 클래스를 보자.

<코드 시작>

코드8

class Criterion(\_Loss):

def \_\_init\_\_(self, way=2, shot=5):

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

self.amount = way \* shot

def forward(self, probs, target, return\_pred\_label=False): # (Q,C) (Q)

target = target[self.amount:]

target\_onehot = torch.zeros\_like(probs)

target\_onehot = target\_onehot.scatter(1, target.reshape(-1, 1), 1)

loss = torch.mean((probs - target\_onehot) \*\* 2)

pred = torch.argmax(probs, dim=1)

acc = torch.sum(target == pred).float() / target.shape[0]

if return\_pred\_label:

return loss, acc, pred, target

else:

return loss, acc

<코드 끝>

[코드8]을 보면 self.amount라는 변수가 있다. 이 변수를 이용해서 target을 슬라이싱한다. target을 슬라이싱하는 이유는 loss를 구할 때 쿼리 데이터만 사용하기 위해서이다.

메타 러닝 모델 학습을 위한 모든 과정을 코드와 함께 설명했다. [코드8]은 앞서 설명한 내용을 바탕으로 실제로 메타 러닝 모델을 학습하는 부분이다.

<코드 시작>

코드8

dev\_interval = 100

best\_acc = -1.0

tbar = tqdm(range(1, 10000))

for episode in tbar:

loss = train(episode)

if episode % dev\_interval == 0:

acc = dev(episode)

if acc > best\_acc:

print('Better acc! Saving model! -> {:.4f}'.format(acc))

best\_acc = acc

tbar.set\_postfix(loss=loss)

1%| | 101/9999 [00:24<1:39:28, 1.66it/s, loss=tensor(0.4911, device='cuda:0', grad\_fn=<MeanBackward0>)]

Better acc! Saving model! -> 0.5374

2%|▏ | 201/9999 [00:49<1:41:05, 1.62it/s, loss=tensor(0.5000, device='cuda:0', grad\_fn=<MeanBackward0>)]

Better acc! Saving model! -> 0.5421

5%|▌ | 500/9999 [02:02<2:03:14, 1.28it/s, loss=tensor(0.4907, device='cuda:0', grad\_fn=<MeanBackward0>)]

Better acc! Saving model! -> 0.5428

32%|███▏ | 3200/9999 [12:54<1:27:25, 1.30it/s, loss=tensor(0.5259, device='cuda:0', grad\_fn=<MeanBackward0>)]

Better acc! Saving model! -> 0.5548

36%|███▌ | 3601/9999 [14:31<1:03:33, 1.68it/s, loss=tensor(0.3047, device='cuda:0', grad\_fn=<MeanBackward0>)]

Better acc! Saving model! -> 0.6022

37%|███▋ | 3701/9999 [14:56<1:05:37, 1.60it/s, loss=tensor(0.4654, device='cuda:0', grad\_fn=<MeanBackward0>)]

Better acc! Saving model! -> 0.6230

39%|███▉ | 3900/9999 [15:44<1:18:19, 1.30it/s, loss=tensor(0.4810, device='cuda:0', grad\_fn=<MeanBackward0>)]

Better acc! Saving model! -> 0.6518

40%|████ | 4000/9999 [16:07<1:18:58, 1.27it/s, loss=tensor(0.2676, device='cuda:0', grad\_fn=<MeanBackward0>)]

Better acc! Saving model! -> 0.6539

41%|████ | 4100/9999 [16:32<1:18:33, 1.25it/s, loss=tensor(0.2664, device='cuda:0', grad\_fn=<MeanBackward0>)]

Better acc! Saving model! -> 0.6652

42%|████▏ | 4200/9999 [16:56<1:16:40, 1.26it/s, loss=tensor(0.2961, device='cuda:0', grad\_fn=<MeanBackward0>)]

Better acc! Saving model! -> 0.6730

43%|████▎ | 4300/9999 [17:20<1:14:57, 1.27it/s, loss=tensor(0.1779, device='cuda:0', grad\_fn=<MeanBackward0>)]

Better acc! Saving model! -> 0.6873

45%|████▌ | 4500/9999 [18:09<1:10:59, 1.29it/s, loss=tensor(0.1996, device='cuda:0', grad\_fn=<MeanBackward0>)]

Better acc! Saving model! -> 0.6883

50%|█████ | 5001/9999 [20:11<50:16, 1.66it/s, loss=tensor(0.1750, device='cuda:0', grad\_fn=<MeanBackward0>)]

Better acc! Saving model! -> 0.6964

51%|█████ | 5100/9999 [20:35<1:05:25, 1.25it/s, loss=tensor(0.3406, device='cuda:0', grad\_fn=<MeanBackward0>)]

Better acc! Saving model! -> 0.7037

59%|█████▉ | 5900/9999 [23:48<53:08, 1.29it/s, loss=tensor(0.0876, device='cuda:0', grad\_fn=<MeanBackward0>)]

Better acc! Saving model! -> 0.7123

60%|██████ | 6000/9999 [24:13<51:30, 1.29it/s, loss=tensor(0.1551, device='cuda:0', grad\_fn=<MeanBackward0>)]

Better acc! Saving model! -> 0.7133

62%|██████▏ | 6200/9999 [25:01<50:09, 1.26it/s, loss=tensor(0.2443, device='cuda:0', grad\_fn=<MeanBackward0>)]

Better acc! Saving model! -> 0.7195

65%|██████▌ | 6501/9999 [26:14<34:55, 1.67it/s, loss=tensor(0.1500, device='cuda:0', grad\_fn=<MeanBackward0>)]

Better acc! Saving model! -> 0.7287

67%|██████▋ | 6700/9999 [27:02<41:29, 1.33it/s, loss=tensor(0.1373, device='cuda:0', grad\_fn=<MeanBackward0>)]

Better acc! Saving model! -> 0.7345

76%|███████▌ | 7600/9999 [30:41<32:03, 1.25it/s, loss=tensor(0.2500, device='cuda:0', grad\_fn=<MeanBackward0>)]

Better acc! Saving model! -> 0.7393

82%|████████▏ | 8200/9999 [33:06<23:03, 1.30it/s, loss=tensor(0.0173, device='cuda:0', grad\_fn=<MeanBackward0>)]

Better acc! Saving model! -> 0.7446

84%|████████▍ | 8400/9999 [33:55<21:26, 1.24it/s, loss=tensor(0.2435, device='cuda:0', grad\_fn=<MeanBackward0>)]

Better acc! Saving model! -> 0.7486

90%|█████████ | 9000/9999 [36:20<13:24, 1.24it/s, loss=tensor(0.0522, device='cuda:0', grad\_fn=<MeanBackward0>)]

Better acc! Saving model! -> 0.7496

92%|█████████▏| 9200/9999 [37:09<10:15, 1.30it/s, loss=tensor(0.0444, device='cuda:0', grad\_fn=<MeanBackward0>)]

Better acc! Saving model! -> 0.7497

93%|█████████▎| 9301/9999 [37:33<07:19, 1.59it/s, loss=tensor(0.2773, device='cuda:0', grad\_fn=<MeanBackward0>)]

Better acc! Saving model! -> 0.7528

94%|█████████▍| 9400/9999 [37:57<07:49, 1.28it/s, loss=tensor(0.0272, device='cuda:0', grad\_fn=<MeanBackward0>)]

Better acc! Saving model! -> 0.7531

98%|█████████▊| 9800/9999 [39:33<02:45, 1.20it/s, loss=tensor(0.2823, device='cuda:0', grad\_fn=<MeanBackward0>)]

Better acc! Saving model! -> 0.7546

99%|█████████▉| 9900/9999 [39:57<01:16, 1.30it/s, loss=tensor(0.2743, device='cuda:0', grad\_fn=<MeanBackward0>)]

Better acc! Saving model! -> 0.7551

100%|██████████| 9999/9999 [40:19<00:00, 4.13it/s, loss=tensor(0.2489, device='cuda:0', grad\_fn=<MeanBackward0>)]

<코드 끝>

[코드8]에 메타 러닝의 학습을 시작하는 코드를 구현했다. 학습 결과를 보면 검증할 때 사용되는 데이터셋은 학습할 때 사용되는 데이터가 아님에도 불구하고 정확도가 향상되는 것을 볼 수 있다.

마지막으로 [코드9]에서는 학습된 메타 러닝 모델을 이용해서 테스트셋에 대한 정확도를 구하는 것을 구현했다.

<코드 시작>

코드9

def test():

model.eval()

correct = 0.

count = 0.

for i in range(100):

data, target = test\_dataloader.get\_batch\_test()

data = data.cuda()

target = target.cuda()

predict = model(data)

\_, acc = criterion(predict, target)

amount = len(target) - support \* 2

correct += acc \* amount

count += amount

acc = correct / count

print('Test Acc: {}'.format(acc))

return acc

>>> test()

Test Acc: 0.6549707651138306

<코드 끝>

[코드9]를 보면 0과 1을 구분하는 이진분류에서 0.7 정도의 정확도를 보여주고 있다. 테스트 데이터셋은 books, dvd, electronics, kitchen\_housewares 등에 대한 데이터셋이다. 이는 학습과정에서는 한번도 사용된 적 없는 데이터이기 때문에 임의로 맞추는 확률인 0.5가 나올것 같지만, 서포트 데이터셋을 통해서 쿼리 데이터셋을 예측하는 구조를 가지고 있기 때문에 0.7정도의 결과를 보이고 있다.

마지막으로 [코드8]에서 학습한 메타 러닝 모델이 제대로 학습된 것인지를 확인하기 위해 서포트 셋을 사용하지 않고(support=0) 학습해보자.

<코드 시작>

코드10

# suppport=0으로 학습한 결과

1%| | 101/9999 [00:17<1:28:42, 1.86it/s, loss=tensor(0.2704, device='cuda:0', grad\_fn=<MeanBackward0>)]

Better acc! Saving model! -> 0.5959

23%|██▎ | 2301/9999 [06:42<1:11:47, 1.79it/s, loss=tensor(0.2603, device='cuda:0', grad\_fn=<MeanBackward0>)]

Better acc! Saving model! -> 0.5967

32%|███▏ | 3201/9999 [09:22<1:04:41, 1.75it/s, loss=tensor(0.2575, device='cuda:0', grad\_fn=<MeanBackward0>)]

Better acc! Saving model! -> 0.5981

41%|████ | 4101/9999 [12:03<59:39, 1.65it/s, loss=tensor(0.2553, device='cuda:0', grad\_fn=<MeanBackward0>)]

Better acc! Saving model! -> 0.5995

100%|██████████| 9999/9999 [29:32<00:00, 5.64it/s, loss=tensor(0.2500, device='cuda:0', grad\_fn=<MeanBackward0>)]

>>> test()

Test Acc: 0.5084260702133179

<코드 끝>

support=0으로 세팅 후 메타 러닝 모델 학습을 진행하면 [코드10]과 같이 정확도가 0.5정도로 나오는 것을 확인했다. 따라서 서포트 데이터셋을 통해서 쿼리 데이터셋을 예측하는 기법, 즉 메타 러닝 기법을 통해서 학습을 위한 학습이 가능하다고 말할 수 있다. support=5로 학습한 모델과 support=0으로 학습한 모델의 confusion matrix를 [표1]을 통해서 확인해보자. 서포트 데이터셋을 사용해서 학습한 모델이 훨씬 좋은 모델임을 알 수 있다.

<표 시작>

표1

# support=5

→ neg/pos

↓ 0/1

[1217 1408]

[ 400 2288]

# support=0

[ 0 3092]

[ 0 3190]

<표 끝>

6.3. GPT2 그리고 GPT3

6.2절에서 다뤘던 메타 러닝 기법은 퓨샷 러닝(Few-shot learning)이다. 서포트 데이터셋 몇 개를 예제로 제공하고(few shot) 그것을 기반으로 쿼리 데이터셋을 맞추는 형태가 퓨샷 러닝이다. 여기에서 서포트 데이터를 한 개만 주는 형태를 원샷 러닝(one-shot learning)이라고 하고 서포트 데이터를 한 개도 주지 않는 형태를 제로샷 러닝(zero-shot learning)이라고 한다. 제로샷 러닝에서는 서포트 데이터셋이 아닌 태스크에 대한 다른 정보가 제공되며, 그 정보를 통해서 결과값을 출력한다. 제로샷 러닝의 개념은 GPT2에서부터 적용되기 시작했다. GPT2 논문에 따르면 어떠한 파라미터나 아키텍쳐의 수정 없이 언어 모델이 제로샷 러닝 세팅에서 동작한다는 것을 보였다고 한다.

<인용 시작>

인용2

We demonstrate language models can perform down-stream tasks in a zero-shot setting – without any parameter or architecture modification

<인용 끝>

[인용2]를 보면 제로샷 세팅에서 여러가지 태스크를 수행할 수 있다는 것을 보였다고 한다. 아쉽게도 GPT2의 구체적인 학습 공식이나 코드는 공개되지 않았기 때문에 GPT2를 학습하는 의사 코드를 구현해보기 힘들다. 이번 절에서는 구현 관점이 아닌, 논문을 이해하는 관점으로 접근해서 GPT2를 이해해보자.

6.3.1. GPT2를 학습하기 위한 접근 방법

GPT2 논문의 Approach 섹션을 보면 일반적인 언어 모델의 공식을 P(output|input)으로 표현했다. 예를 들어서 텍스트 분류 모델을 학습했을 경우 주어진 입력 값에 대해서 positive=1, negative=0를 출력할 확률을 구한다. GPT2에서는 한 발자국 더 나아가서 입력 값에 테스크 정보를 더한다. 즉, P(output|input,task) 형태로 언어 모델 공식을 정했다. 여기에서 태스크는 여러 가지 종류의 자연어 처리 태스크를 의미한다. 예를 들어 텍스트 분류, 기계 번역, 질의 응답 등등이 될 수 있다. 이렇게 테스크의 정보를 입력의 또 다른 조건 값으로 제공하는 것을 태스크 컨디셔닝이라고 소개하고 있고, 태스크를 제공하는 방법은 자유로운 방식으로 가능하다고 설명하고 있다.

<블록 시작>

블록2

기계어 번역 태스크: (translate to french, english text, french text)

질의응답 태스크: (answer the question, document, question, answer)

<블록 끝>

[블록2]를 보면 기계어 번역 태스크의 경우, 입력 값은 "translate to french"라는 자연어와 번역을 할 언어(english text) 두 개 제공된다. 질의응답 태스크의 경우 "answer the question"이라는 자연어와 document, question 등의 정보가 입력으로 제공되고 그것을 이용해서 answer를 추론하게 된다.

6.3.2. GPT2의 학습 데이터셋과 멀티태스크

GPT2를 학습한 데이터셋에 대해서 이해해보자. GPT2는 WebText 데이터셋으로 학습됐다. WebText 데이터셋은 Reddit이라는 소셜 미디어 플랫폼에서 아웃바운드 링크만을 크롤링해서 최소 3 karma 이상의 정보만 사용해서 만든 데이터셋이다. [## 각주1] GPT2는 Reddit의 최소 3karma 이상의 데이터셋을 약 40GB의 모아서 학습된 모델이다.

## 각주1

Reddit의 karma는 정보가 다른 사람에게 얼마나 유용했는지를 나타내는 지표로 해석하면 된다.

GPT2를 사전학습할 때 멀티태스크를 학습할 수 있도록 하기 위해 in-context learning이라는 개념을 사용했다. GPT3 논문의 Introduction에 보면 [인용3]과 같이 언급하고 있다.

<인용 시작>

인용3

Recent work [RWC+19]

attempts to do this via what we call “in-context learning”, using the text input of a pretrained language model as a form

of task specification: the model is conditioned on a natural language instruction and/or a few demonstrations of the task

and is then expected to complete further instances of the task simply by predicting what comes next.

<인용 끝>

[인용3]에서 RWC+19로 참조하고 있는 논문이 GPT2 논문이다. GPT2논문에서 in-context learning이라는 개념을 통해서 태스크 컨디셔닝을 할 수 있게 됐고, 태스크 컨디셔닝은 텍스트 입력을 통해서 주어진다고 한다. 태스크는 사람이 이해할 수 있는 자연어 형태로 주어지며, 모델은 단순히 다음에 올 단어가 무엇인지를 맞추는 방식으로 태스크를 수행하게 된다.

GPT3논문에서 in-context learning 개념을 [그림4]을 통해서 설명하고 있다.

<그림 시작>

그림4

GPT3에 있는 in-context learning 그림

https://arxiv.org/pdf/2005.14165.pdf

영어-불어 변환

영화 리뷰 분류

<그림 끝>

[그림4]를 보자. 전체적으로는 outer-loop를 통해서 다음에 올 단어가 무엇인지를 맞추는 언어 모델링을 학습하는데, 각 시퀀스마다 in-context learning을 학습하고 있다. in-context learning은 "5 + 8 = 13", "7 + 2 = 9"와 같이 데이터와 정답의 쌍으로 이루어진 셋을 여러 개 이어서 패턴을 인식할 수 있는 능력을 학습시키는 과정이다.

<블록 시작>

블록3

# [그림4]의 sequence#1 패턴

숫자 + 숫자 = 숫자

# [그림4]의 sequence#2 패턴

단어 => 단어

# [그림4]의 sequence#3 패턴

단어 => 단어

<블록 끝>

[블록3]에서 각 시퀀스마다 학습되는 패턴을 설명하고 있다. [그림4]의 sequence#1은 숫자와 숫자를 더하기(+)한 후 숫자가 나오는 패턴을 계속 반복해서 학습한다는 것을 보여주고 있고, [그림4]의 sequence#2는 단어와 단어 사이를 =>로 구분해서 연결하는 패턴을 계속 반복해서 학습한다는 것을 보여주고 있다.

6.3.3. GPT2 성능평가 결과

GPT2의 성능을 공개된 8개의 데이터셋을 이용해서 평가했다. 그 결과는 [표2]과 같다.

<표 시작>

표2

chapter6 엑셀 참고

https://openai.com/blog/better-language-models/

<표 끝>

[표2]의 결과는 Fine-Tuning 없이 사전학습만 수행한 모델로 평가한 결과이다. 이처럼 특별한 사전학습 없이 특정 태스크를 수행할 수 있게 됐다는 것이 GPT2 모델의 가장 중요한 성과이다. 다음 절에서는 실제 소스코드를 이용해서 GPT2를 통해서 문장 생성하는 예제를 공부해보자.

6.3.4. GP2를 통한 문장 생성

GPT2의 사전학습은 이전 토큰들을 통해 다음 토큰을 맞추는 언어 모델링이다. 이번 절에서는 트랜스포머의 GPT2를 이용해서 문장을 생성해보려고 한다.

<블록 시작>

블록4

Whether the Ministry of Gender Equality & Family should remain in existence again surfaced as a hot button issue Friday

# http://www.koreaherald.com/view.php?ud=20211022000603

<블록 끝>

[블록4]의 문장은 코리아 헤럴드 기사의 일부이다. GPT2를 통해서 그 다음 문장들을 생성해보자.

<코드 시작>

코드11

MODEL\_CLASSES = {

'gpt2': (GPT2LMHeadModel, GPT2Tokenizer),

}

model\_type = 'gpt2'

model\_name\_or\_path = 'gpt2'

model\_class, tokenizer\_class = MODEL\_CLASSES[model\_type]

tokenizer = tokenizer\_class.from\_pretrained(model\_name\_or\_path)

model = model\_class.from\_pretrained(model\_name\_or\_path)

model.to(device)

model.eval()

<코드 끝>

[코드11]은 GPT2 모델을 로딩하는 코드이다. 전체 코드는 이 책의 레포지토리에서 chapter6/gpt2-generation-example.ipynb를 참고하면 된다.

<코드 시작>

코드12

prompt = f"Whether the Ministry of Gender Equality & Family should remain in existence again surfaced as a hot button issue Friday "

text = generate(model, tokenizer, prompt)

>>> prompt + text

'Whether the Ministry of Gender Equality & Family should remain in existence again surfaced as a hot button issue Friday ・Due to ongoing problems at the Ministry in the area of the institution under which the Paternity Service'

## ・Due to ongoing problems 부터 끝까지는 굵은 글씨로 부탁드립니다.

<코드 끝>

로딩된 모델을 바로 이용해서 [코드12]와 같이 문장을 생성해보자. [블록3]에 있는 텍스트를 프롬프트(prompt)라는 변수에 저장했다. GPT2에서 프롬프트는 GPT2에게 주어진 입력 문장을 의미한다. GPT2는 다음 단어를 생성하도록 사전학습됐기 떄문에 프롬프트 이후에 알맞는 단어들을 생성하게 된다. [코드12]에서 사용한 generate함수는 chapter6/generation.py를 참고하면 된다. 다음 절에서는 이번 절의 소스코드를 재사용하여 GPT2의 퓨샷 러닝 성능을 확인해보자.

6.3.5. GPT2를 이용한 퓨샷 러닝

사전학습된 GPT2를 이용해서 퓨샷 러닝을 해보자. 파인튜닝 없이 사전학습된 GPT2를 로딩한 후 퓨샷 세팅으로 간단한 실행을 해보자. 이 절의 소스코드는 레포지토리의 chapter6/gpt2-verb-example.ipynb를 참고하면 된다. 이 절에서는 예제 소스코드를 통해서 GPT2가 패턴을 이해하고 있다는 것을 확인해볼 것이다. 사용할 프롬프트는 [블록5]과 같다.

<블록 시작>

블록5

play = played . sing = sang . view = viewed . act = acted . say = said . type = typed . note = noted . see = saw . clean = cleaned . tell = told . click =

<블록 끝>

[블록5]에서의 태스크는 영어 단어의 원형과 과거형의 나열이다. [블록5]를 자세히 보면 원형과 과거형을 =으로 연결했고, "원형 = 과거형"셋을 .을 통해 여러개 이은 형태를 하고 있다. 그리고 마지막은 "click ="로 끝났다. 이는 GPT2에게 다음과 같이 물어보는 것과 같다. "이봐 GPT2!! play의 과거형이 played고 ... tell의 과거형은 told야. 그러면 click의 과거형은 뭐니?". GPT2는 기본적으로 주어진 프롬프트 이후의 문장을 생성하도록 학습돼 있다. 제대로된 결과를 얻는다면 clicked가 생성돼야 한다. [블록5]와 같이 패턴이 포함된 프롬프트를 입력해도 원하는 결과를 얻을 수 있게 하는 것이 GPT2의 핵심이다.

코델로딩은 [코드11]과 같이 하면 되고 문장을 생성하는데 사용하는 generate 함수 역시 chapter6/generation.py를 사용하면 된다.

이제 입력 값을 준비해보자. [코드12]를 통해 five\_verbs.txt를 읽어서 텍스트 리스트 형태로 입력 값을 준비해보자.

<코드 시작>

코드12

verbs = []

with open('five\_verbs.txt', 'r') as f:

for line in f:

verbs.append(line.lower().replace('\n',''))

>>> verbs

['click', 'work', 'walk', 'run', 'jump']

<코드 끝>

그 다음은 [코드12]에서 읽은 file\_verbs.txt의 각 단어를 프롬프트의 마지막 끝에 입력해서 generate 함수에 넣어주면 된다. 이번 절에서는 10샷으로 설정했기 때문에 "원형 = 과거형"으로 이루어진 10개의 문장이 서포트셋으로 사용된다. 서포트셋과 최종적으로 사용할 프롬프트는 [블록5]를 통해 확인해보자.

<블록 시작>

블록6

# 10개의 서포트셋

play = played # 1 shot

sing = sang # 2 shot

view = viewed # 3 shot

act = acted # 4 shot

say = said # 5 shot

type = typed # 6 shot

note = noted # 7 shot

see = saw # 8 shot

clean = cleaned # 9 shot

tell = told # 10 shot

# 최종적으로 사용할 프롬프트 → 각 프롬프트에 대해서 generate 함수를 실행해서 clicked, worked 등의 결과를 얻을 수 있다.

play = played . sing = sang . view = viewed . act = acted . say = said . type = typed . note = noted . see = saw . clean = cleaned . tell = told . click =

play = played . sing = sang . view = viewed . act = acted . say = said . type = typed . note = noted . see = saw . clean = cleaned . tell = told . work =

play = played . sing = sang . view = viewed . act = acted . say = said . type = typed . note = noted . see = saw . clean = cleaned . tell = told . walk =

play = played . sing = sang . view = viewed . act = acted . say = said . type = typed . note = noted . see = saw . clean = cleaned . tell = told . run =

play = played . sing = sang . view = viewed . act = acted . say = said . type = typed . note = noted . see = saw . clean = cleaned . tell = told . jump =

<블록 끝>

마지막으로 [코드13]을 통해서 [블록6]에서 준비한 5개의 프롬프트에 대한 결과를 확인해보자.

<코드 시작>

코드13

plurals = []

for verb in verbs:

prompt = f"play = played . sing = sang . view = viewed . act = acted . say = said . type = typed . note = noted . see = saw . clean = cleaned . tell = told . {verb} ="

plural = generate(model, tokenizer, prompt)

plural = plural.split(".")[0]

plurals.append(plural)

>>> for v, p in zip(verbs, plurals):

>>> print(f'{v} -> {p}')

click -> clicked

work -> worked

walk -> walked

run -> run

jump -> jump

<코드 끝>

마지막 jump를 제외하면 단어의 원형과 과거형이 잘 매칭된 것을 확인할 수 있다.

6장에서는 GPT2에 대해서 알아봤다. GPT2와 GPT3는 다음 단어를 맞추는 방식으로 학습된 언어 모델이다. 5장에서 알아본 BERT 또는 그 외의 모델과의 가장 큰 차이점은 퓨샷 러닝 기법의 적용이다. GPT2나 GPT3에서는 파인튜닝 없이 사전학습된 모델만으로 여러가지 태스크를 실행할 수 있다는 것을 알았고, 많은 경우 SOTA를 달성하기도 했다. 6장의 마지막 절에서는 트랜스포머의 GPT2를 사용해서 문장 생성하는 것을 알아봤고, 프롬프트 내에 패턴이 있을 경우 그 패턴에 맞는 정답을 맞춰낼 수 있다는 것도 알아봤다.