Chapter_05_요약_text-generation.ipynb

transformer · 2024. 7. 31. 16:41

출처

https://product.kyobobook.co.kr/detail/S000200330771

아래그림)

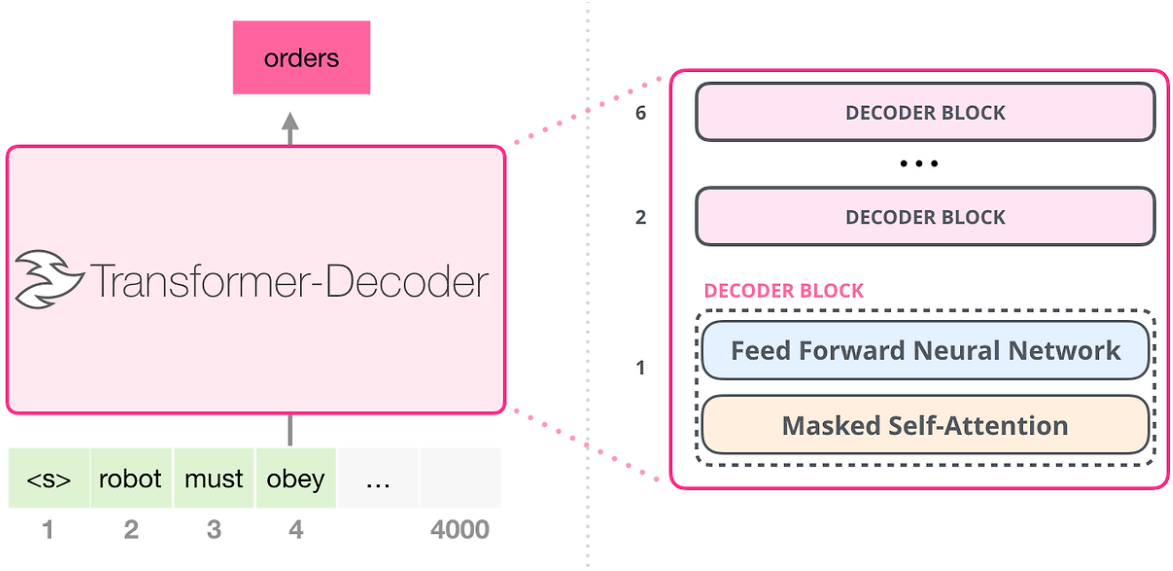
causal launguage model

앞에 단어를 모두 보고 그다음 단어를 맞추는 방법(그 확률값들을 구해서 다 곱해주는것)

$$x \qquad y = y_1, y_2 \cdots y_t$$

$$P(y|x) = P(y_1|y_1, x) \cdot P(y_3|y_1, y_2, x) \cdots$$

$$= \prod_{t=1}^{N} P(y_t | y_{<t}, x)$$



출처

https://chloamme.github.io/2021/12/08/illustrated-gpt2-korean.html


```
import pandas as pd

input_txt = "Transformers are the" #input값
tokenized = tokenizer(input_txt, return_tensors="pt") # 위의
input_txt를 쪼개고 몇번단어인지 숫자로 변환 attention_mask는 일단 신경안써도됨
tokenized
```

```
tokenizer.decode(tokenized['input_ids'][0]) # tokenizer의 decode
(tokenizer의 encoding은 문자를  input_txt를 쪼개는거고 숫자로 변환,
tokenizer의 decode는 변환된 숫자를 다시 문자로 변환하고 붙이기까지함)
```

```
output.logits.shape # 4는 위의 'input_ids': tensor([[41762,   364,
389,   262]] 에서 sequance길이, 1은 batch size , 50257은 vocab
size(data갯수와는 전혀다름) vocab size는 데이터 안의 단어갯수, data는 문장임 1
문장이거나 여러문장이 data, vocab size는 중복은 제외되고 생각(단어갯수아니고
token개수 영어면 알파벳으로 다 자른후 2개씩 묶었을때 가장많이 나오는 것 확인. 이것을
2개묶은 한쌍을 vocab에 추가, 이것을 원하는 vocab size나올때까지 반복. 자주나오는
쌍들이 저장됨 몇만개까지, 4는 결과값까지 포함해서 [41762,   364,   389,   262]
다음 하나가 결과라서 4
```

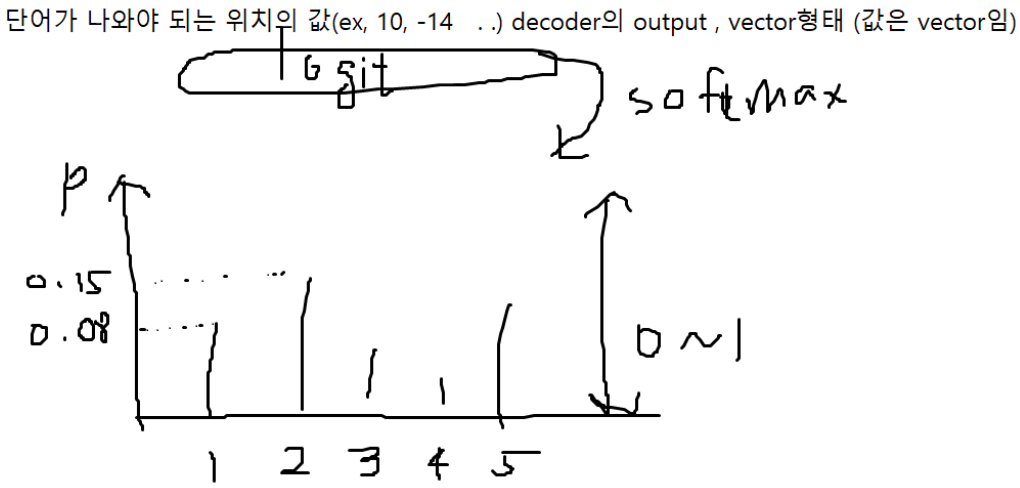
```
put_ids = tokenizer(input_txt, return_tensors="pt")
["input_ids"].to(device) #input값을 tokenizing
iterations = []
n_steps = 8
choices_per_step = 5

with torch.no_grad():# 사전학습된 모델을 갖고와서 추가적인 학습없이 추론만함 no
gradient해서 빨리코드가 돌아감
    for _ in range(n_steps):
        iteration = dict()
        iteration["Input"] = tokenizer.decode(input_ids[0])
        output = model(input_ids=input_ids)
        # 첫 번째 배치의 마지막 토큰의 로짓을 선택해 소프트맥스를 적용합니다.
        next_token_logits = output.logits[0, -1, :] # 0은 batch size
가 1이라서 리스트처럼 첫번째것만 갖고오고, -1은 sequence인 "Transformers are
the" 다음의 단어를 갖고오기위해 마지막단어인 -1인 the까지 input했을때 뭐를 결과로
내뱄는지 output.logits를 실행
        next_token_probs = torch.softmax(next_token_logits, dim=-1)
        sorted_ids = torch.argsort(next_token_probs, dim=-1,
descending=True) #softmax한값중 가장 높은값부터 정렬
        # 가장 높은 확률의 토큰을 저장합니다.
        for choice_idx in range(choices_per_step): # 상위 다섯개를 추가해
줌
            token_id = sorted_ids[choice_idx]
            token_prob = next_token_probs[token_id].cpu().numpy()
            token_choice = (
```

```
                f"{tokenizer.decode(token_id)} ({100 *
token_prob:.2f}%)"
            )
            iteration[f"Choice {choice_idx+1}"] = token_choice
        # 예측한 다음 토큰을 입력에 추가합니다.
        input_ids = torch.cat([input_ids, sorted_ids[None, 0,
None]], dim=-1)
        iterations.append(iteration)

pd.DataFrame(iterations)
```

logit 정의



출처: https://haje01.github.io/2019/11/19/logit.html

로짓(Logit) 이란?

먼저, 오즈(odds)

odds는 실패비율 대비 성공비율을 설명하는 것.
어떤 이벤트가 15번 시행 중 5번 성공했을 때, 성공 비율은 515515이고 실패 비율은 10151015이다. 이때 오즈는: 5151015=5105151015=510
같은 식으로, 확률 p에 대한 오즈는 다음과 같이 정의된다:

p1−pp1−p

로짓

확률 p의 로짓 L은 다음과 같이 정의된다:

L=lnp1−pL=lnp1−p

즉, 오즈에 자연로그를 씌운 것. 로짓(logit)은 log + odds에서 나온 말.

오즈는 그 값이 1보다 큰지가 결정의 기준이고, 로짓은 0보다 큰지가 결정의 기준.

이것의 역함수는:

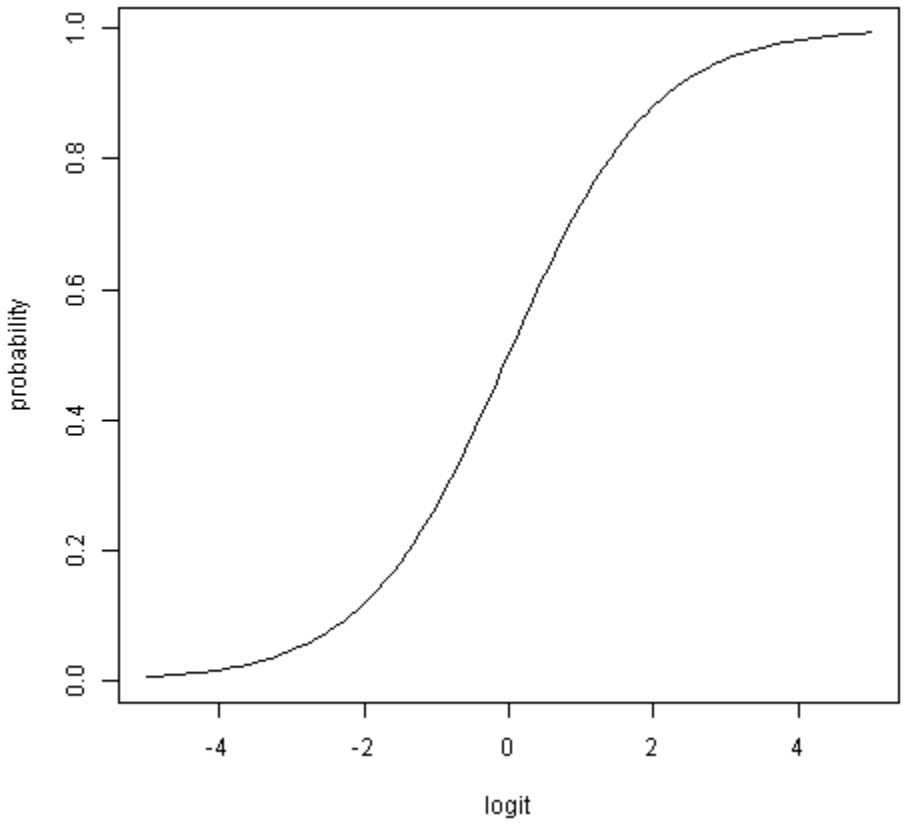
p=11+e−Lp=11+e−L

아래 위로 e−Le−L을 곱해주면 아래와 같이 시그모이드 함수로 나온다:

p=e−Le−L+1p=e−Le−L+1

확률의 범위는 [0,1][0,1] 이나, 로짓의 범위는 [−∞,∞][−∞,∞]이다.

아래 그림은 확률과 로짓의 비선형 관계를 보여줌:



로짓의 필요성: 데이터를 두 그룹으로 분류하는 문제

기본적인 방법은 로지스틱 회귀분석으로, 종속변수 y가 0 또는 1을 갖기에, 단순 선형 함수 y=wx+by=wx+b로는 풀기가 힘들다(입력값이 커지면 출력값의 해석이 곤란).

확률 p의 범위는 [0,1][0,1], Odds(p)의 범위는 [0,∞][0,∞], log(Odds(p))는 [−∞,∞][−∞,∞]가 되어, 다음과 같은 형태로는 선형분석이 가능하다.

log(Odds(p))=wx+blog(Odds(p))=wx+b

위의 식을 잘 설명하는 시그모이드 함수의 w와 b를 찾는 문제로 바꾼다.

딥러닝에서 로짓

[0,1][0,1] 범위인 확률을 [−∞,∞][−∞,∞] 범위로 넓히는 로짓의 특성때문에, 딥러닝에서는 확률화되지 않은 날 예측 결과를 로짓이라고 부른다. 멀티 클래스 분류문제에서 보통 softmax 함수의 입력으로 사용된다.

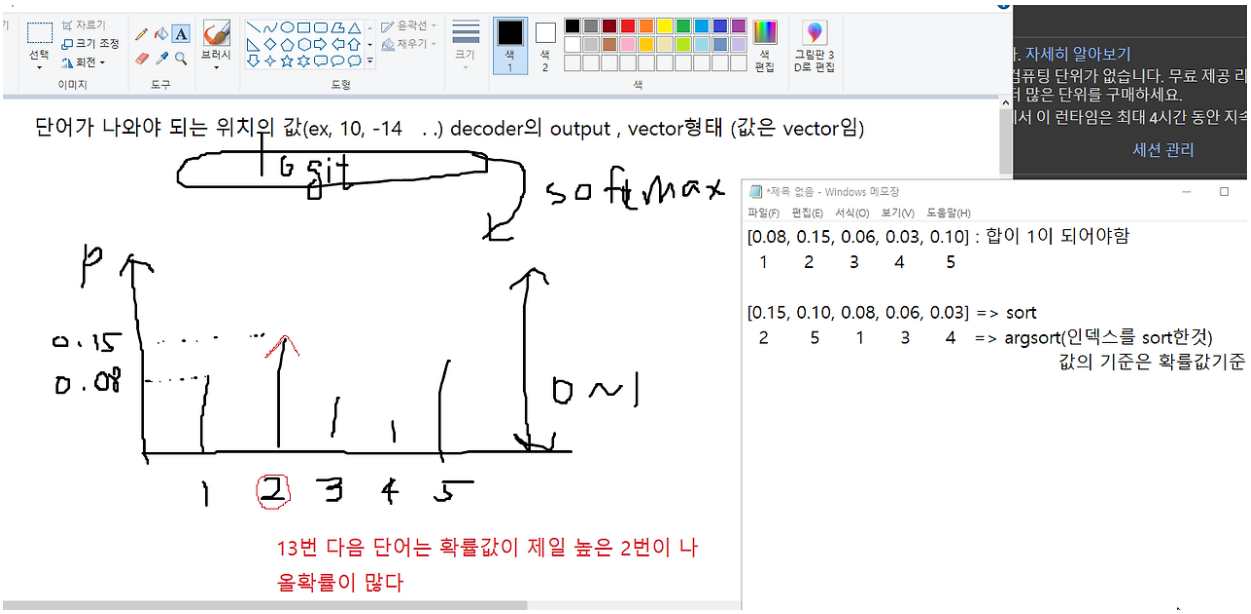
아래그림) [0.08, 0.15, 0.06, 0.03, 0.10] : 합이 1이 되어야함

1 2 3 4 5

[0.15, 0.10, 0.08, 0.06, 0.03] => sort

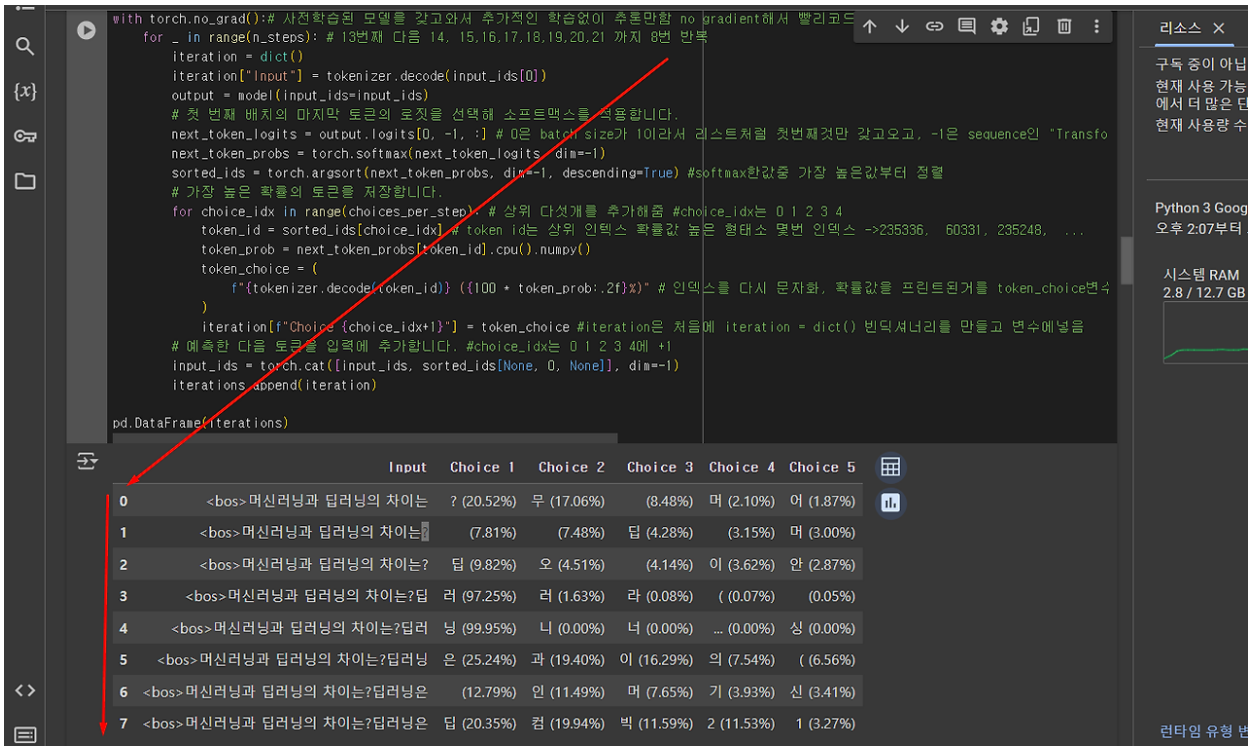
2 5 1 3 4 => argsort(인덱스를 sort한것)

값의 기준은 확률값기준



아래그림) # 13번째 다음 14, 15,16,17,18,19,20,21 까지 8번 반복

확률값 높은 5개 사전학습된 모델을 갖고와서 추가적인 학습없이 추론만함 no gradient해서 빨리코드가 돌아감



	Input	Choice 1	Choice 2	Choice 3	Choice 4	Choice 5
0	<bos>머신러닝과 딥러닝의 차이는	? (20.52%)	무 (17.06%)	(8.48%)	머 (2.10%)	어 (1.87%)
1	<bos>머신러닝과 딥러닝의 차이는?	(7.81%)	(7.48%)	딥 (4.28%)	(3.15%)	머 (3.00%)
2	<bos>머신러닝과 딥러닝의 차이는?	딥 (9.82%)	오 (4.51%)	(4.14%)	이 (3.62%)	안 (2.87%)
3	<bos>머신러닝과 딥러닝의 차이는?딥	러 (97.25%)	러 (1.63%)	라 (0.08%)	((0.07%)	(0.05%)
4	<bos>머신러닝과 딥러닝의 차이는?딥러	닝 (99.95%)	니 (0.00%)	너 (0.00%)	… (0.00%)	싱 (0.00%)
5	<bos>머신러닝과 딥러닝의 차이는?딥러닝	은 (25.24%)	과 (19.40%)	이 (16.29%)	의 (7.54%)	((6.56%)
6	<bos>머신러닝과 딥러닝의 차이는?딥러닝은	(12.79%)	인 (11.49%)	머 (7.65%)	기 (3.93%)	신 (3.41%)
7	<bos>머신러닝과 딥러닝의 차이는?딥러닝은	딥 (20.35%)	컴 (19.94%)	빅 (11.59%)	2 (11.53%)	1 (3.27%)

```
# input_ids = tokenizer(input_txt, return_tensors="pt")
["input_ids"].to(device) #input값을 tokenizing
iterations = []
n_steps = 8
choices_per_step = 5

with torch.no_grad():# 사전학습된 모델을 갖고와서 추가적인 학습없이 추론만함 no
gradient해서 빨리코드가 돌아감
    for _ in range(n_steps): # 13번째 다음 14, 15,16,17,18,19,20,21 까
지 8번 반복
        iteration = dict()
        iteration["Input"] = tokenizer.decode(input_ids[0])
        output = model(input_ids=input_ids)
        # 첫 번째 배치의 마지막 토큰의 로짓을 선택해 소프트맥스를 적용합니다.
        next_token_logits = output.logits[0, -1, :] # 0은 batch size
가 1이라서 리스트처럼 첫번째것만 갖고오고, -1은 sequence인 "Transformers are
the" 다음의 단어를 갖고오기위해 마지막단어인 -1인 the까지 input했을때 뭐를 결과로
내뱄는지 output.logits를 실행
        next_token_probs = torch.softmax(next_token_logits, dim=-1)
        sorted_ids = torch.argsort(next_token_probs, dim=-1,
descending=True) #softmax한값중 가장 높은값부터 정렬
        # 가장 높은 확률의 토큰을 저장합니다.
        for choice_idx in range(choices_per_step): # 상위 다섯개를 추가해
```

```
줌 #choice_idx는 0 1 2 3 4
            token_id = sorted_ids[choice_idx] # token id는 상위 인덱스
확률값 높은 형태소 몇번 인덱스 ->235336,  60331, 235248,  ...
            token_prob = next_token_probs[token_id].cpu().numpy()
            token_choice = (
                f"{tokenizer.decode(token_id)} ({100 *
token_prob:.2f}%)" # 인덱스를 다시 문자화, 확률값을 프린트된거를 token_choice
변수에 넣는다.
            )
            iteration[f"Choice {choice_idx+1}"] = token_choice
#iteration은 처음에 iteration = dict() 빈딕셔너리를 만들고 변수에넣음
        # 예측한 다음 토큰을 입력에 추가합니다. #choice_idx는 0 1 2 3 4에 +1
        input_ids = torch.cat([input_ids, sorted_ids[None, 0,
None]], dim=-1)
        iterations.append(iteration)

pd.DataFrame(iterations)
```

아래그림 greedy decoding으로 (확률값제일높은것먼저) generate(max_new_token)해서 다음단어 나올것 8개 생성

```
input_ids = tokenizer(input_txt, return_tensors="pt")
["input_ids"].to(device) #숫자 제일 높은확률값 greedy decoding
output = model.generate(input_ids, max_new_tokens=n_steps,
do_sample=False) #확률값높은것 1개, #머신러닝과 딥러닝의 차이는이 input_ids,
                            # generate(input_ids)가 다음 단어 나올 제일 높
은 확률값단어를 붙여쓴다.
                            # max_new_tokens=n_steps 는 8번반복해서 토큰생
성 ,  do_sample=False은 무작위 추출인 sampling이 없다.
output.shape    # 밑에 결과값, 1은 batch size 1, 21은 sequence길이, 원래
input 13+ 새로생성된길이 8더해서 21나옴
```

greedy choice

7 5 3 1

50000 * 7

10000 * 5

5000 * 3

1000 * 1

아래그림) beam search

greedy로 나오는 값보다 다음단어를 곱해서 확률값이 큰값들을 n개씩(빔크기) 곱해서 후보로 생각하는것이 빔서치

transfomers - are 0.6 - a 0.1

- the 0.5

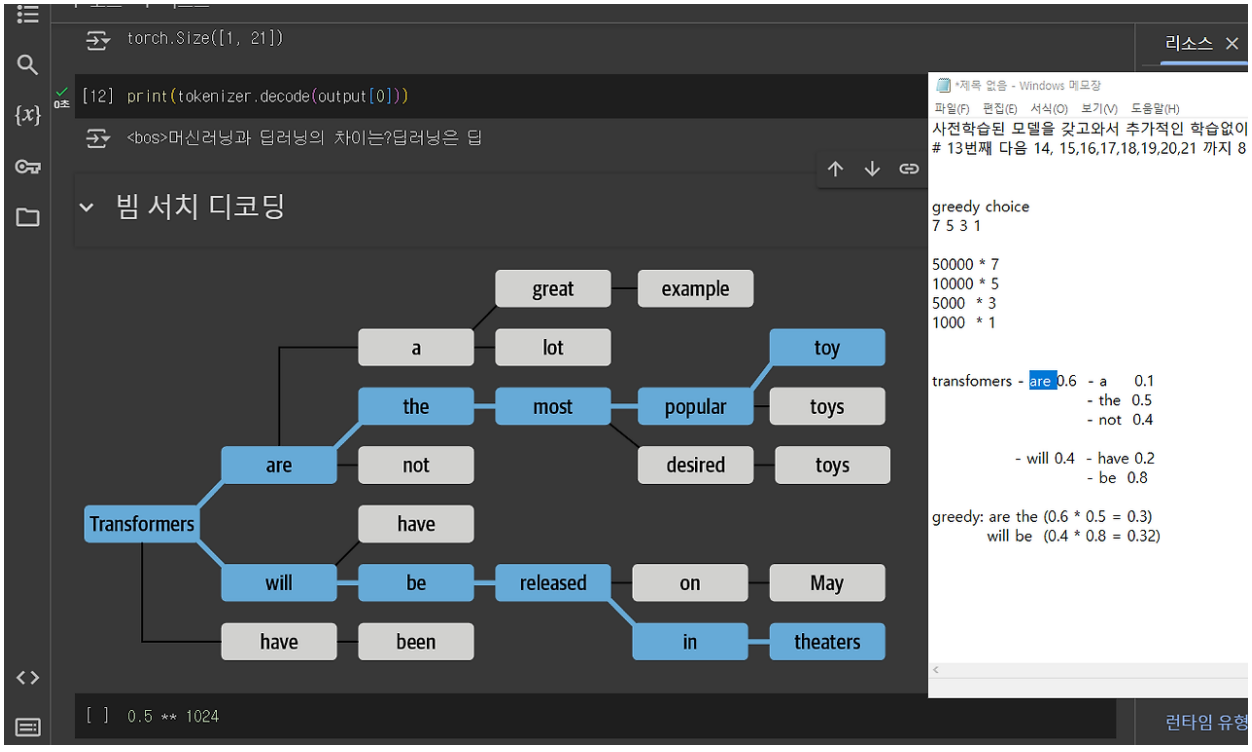
- not 0.4

- will 0.4 - have 0.2

- be 0.8

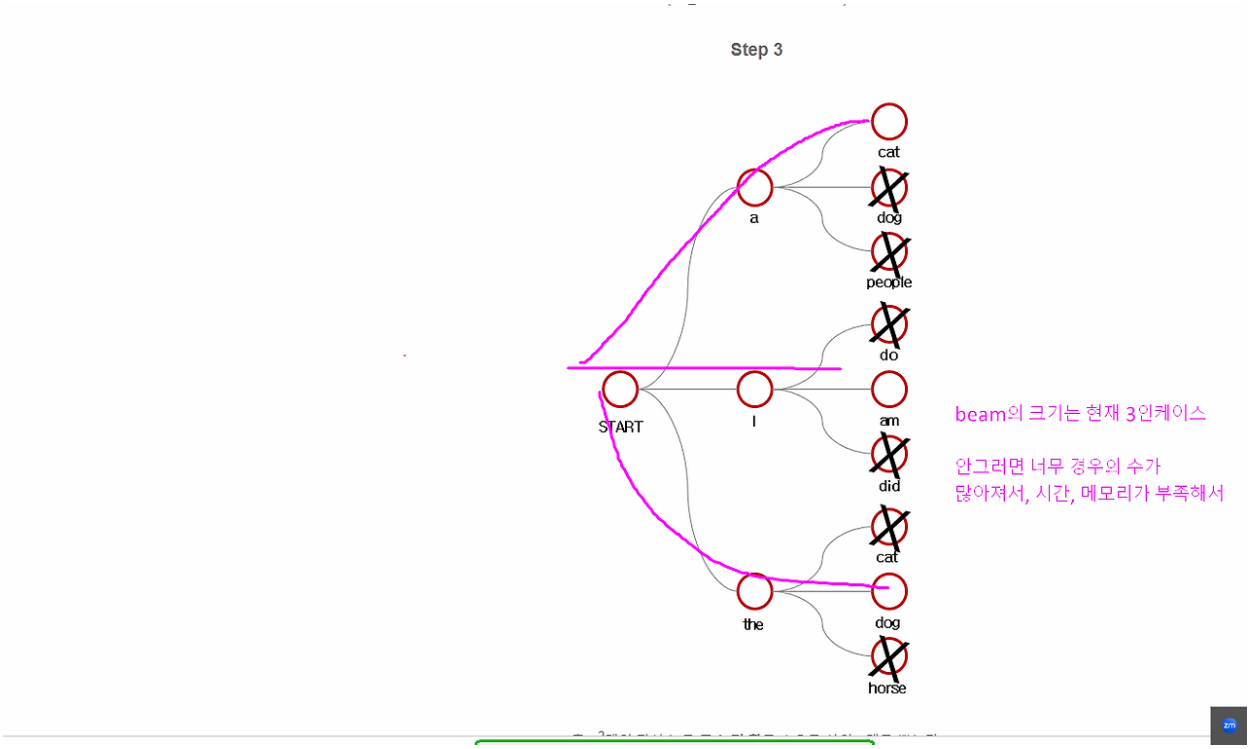
greedy: are the (0.6 * 0.5 = 0.3)

will be (0.4 * 0.8 = 0.32)



아래그림) beam의 크기는 현재 3인케이스

안그러면 너무 경우의 수가
많아져서, 시간, 메모리가 부족해서



$10^{0} = 100$

$\log_{10} 100 = 2$

$\log_{10} 1000 = 3$

$\log_{10} 10^2 = 2$

$10^2 \times 10^3 = 10^5$

$\log_{10} 10^5 = 5$

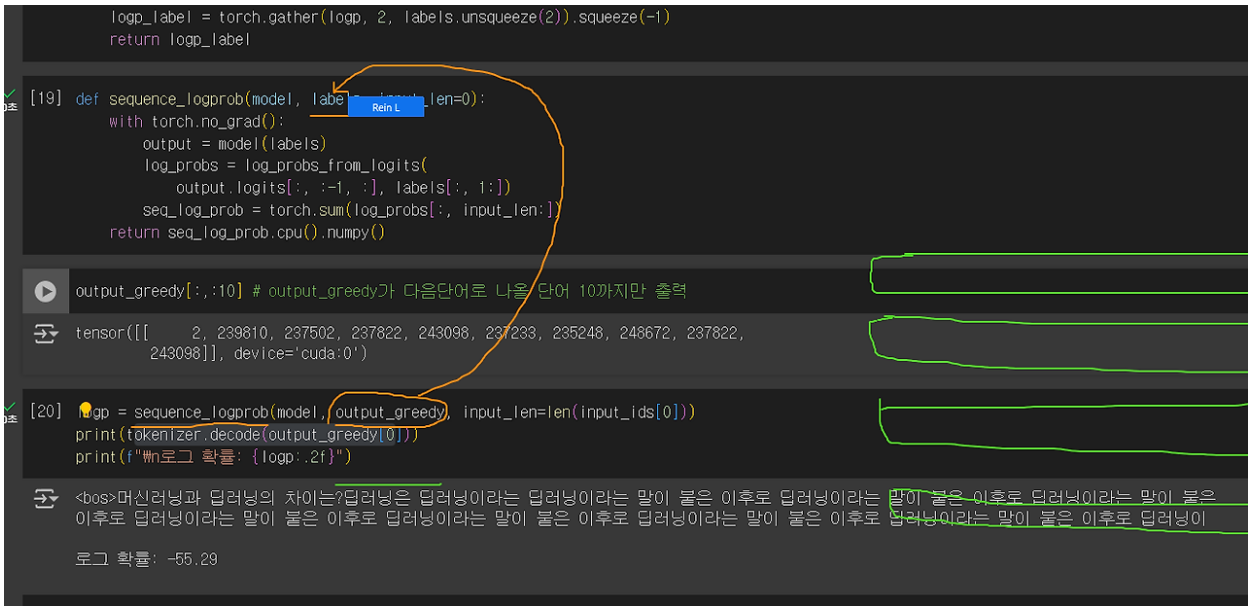
$\log_{10} 10^2 = 2 + \log_{10} 10^3 = 3$

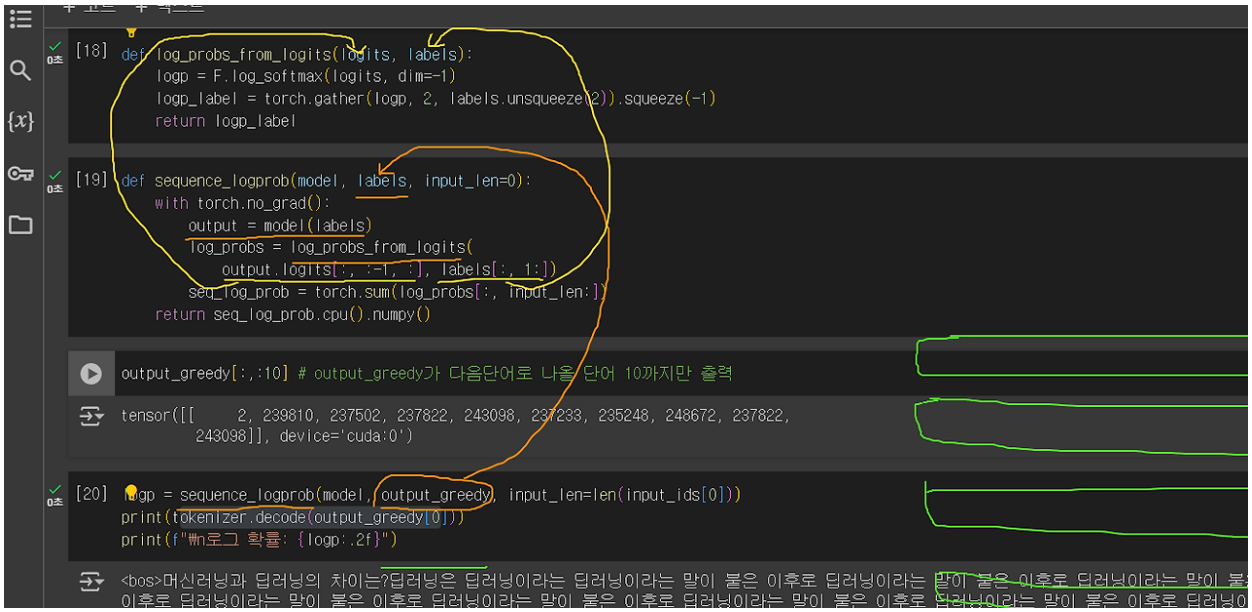
$\log_{10} 10^5 = 5$

$10^2 \times 10^3 = 10^5$

$\log(AB) = \log A + \log B$

$\log 10^2$





아래그림) text generation 정리

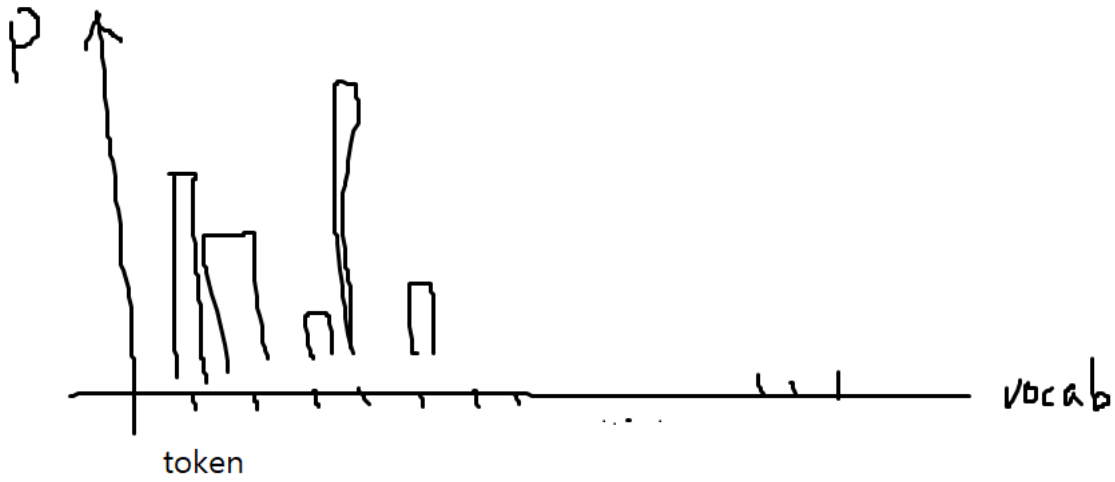
한스텝씩 각위치에서 output token 을 결정- 입력으로 주어진 단어 다음이 뭔지를 구하는것- 하나구하고, 다음위치에서 또 구함

방법 1. greedy decoding

- 각위치에서 확률값 가장 높은것을 정함

방법 2. beam search

- 상수k 하나를 정해서, 그 갯수 만큼의 후보 sequence(동시등장 확률이 가장 높은값) 를 갖는다.



greedy decoding과 beam search

greedy(k=1) - 밥을(확률이 큰거를 정하고 다음으로 넘어감), 그다음 먹었다 를 결정
beam search(k=2) 는 k=2로 후보값큰것을 2개정하고 다음 token에서도 2개정하고 곱하는것을 반복
그러나 컴퓨터가 소숫점 작은수를 계산을 정확하게 못해서 log함수를 쓴다.

나는

sequence는 여러개를 순서있게 나열

첫번째 두번째
token token
-- 컴퓨터가 소숫점 작은수를 계산을 정확하게 못해서 log함수를 쓴다.
밥을 0.6 - 먹었다 0.3 => 0.18
지었다 0.2 => 0.12
..
..

학교에 0.4 - 간다 0.5 => 0.2

도착했다 0.3 => 0.12

...

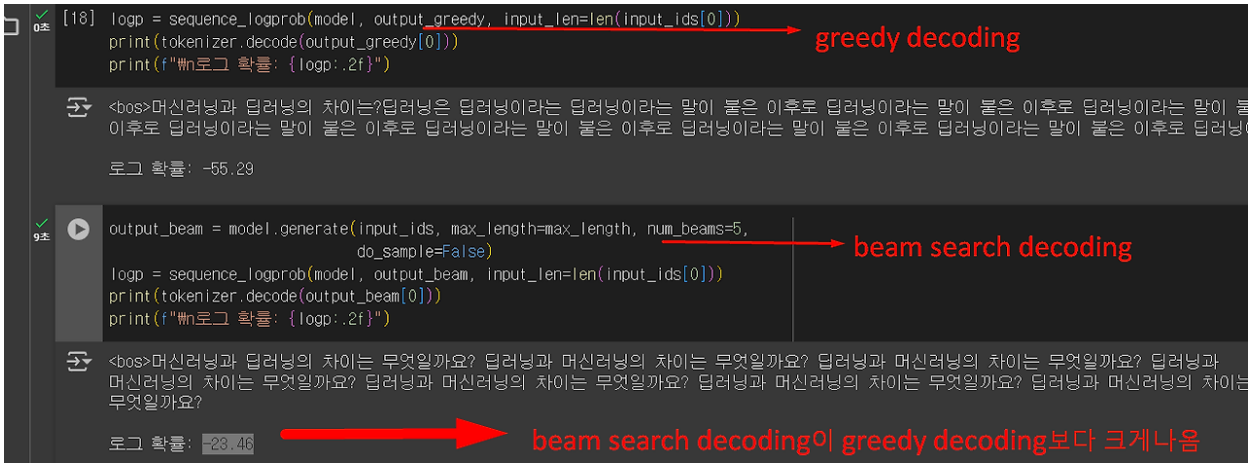
운동을 0.1 -

..

..

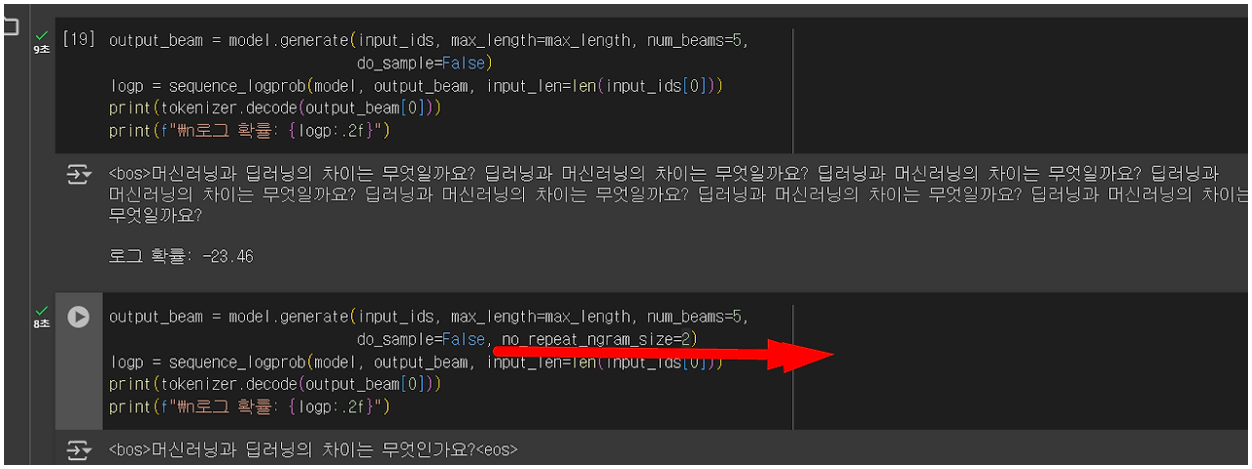
.

log함수에서 beam searching decoding이 greedy decoding 보다 확률이 크게나옴



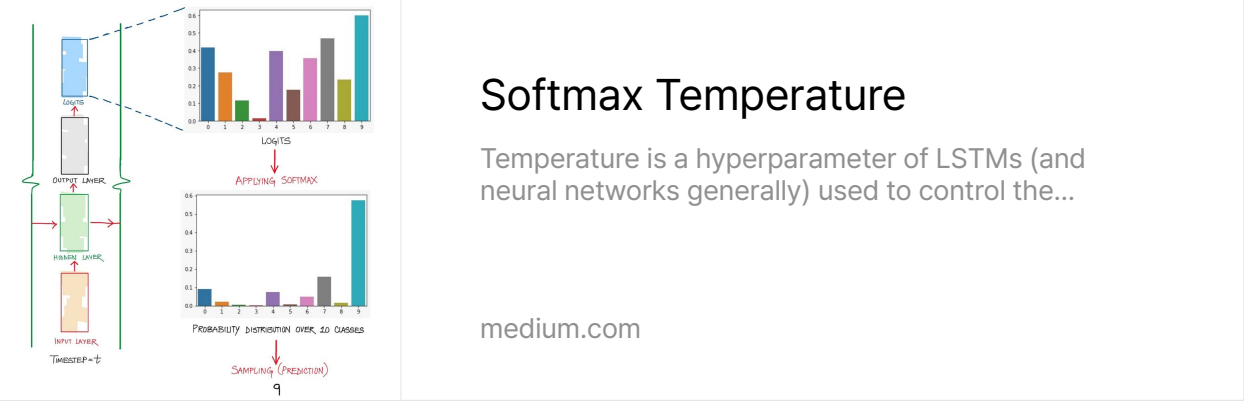
똑같은 token2개 반복안되게 함

그러나 항상도움이 되는거는 아님(ex. new york 이 한번나오면 2번나올수 없어서 new york의 자세한 문장생성이 어려움)



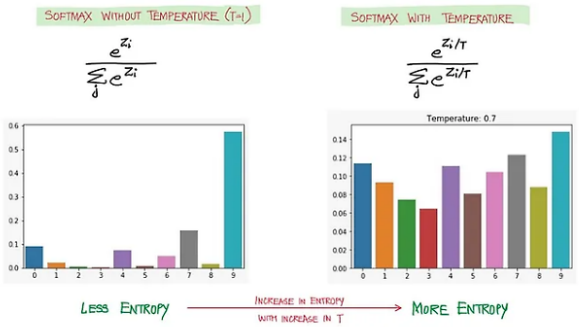
아래그림) temperature: token의 확률값의 상대적인 크기를 줄인다.(항상줄이는건 아니고, 더 크게 할수도 있음)

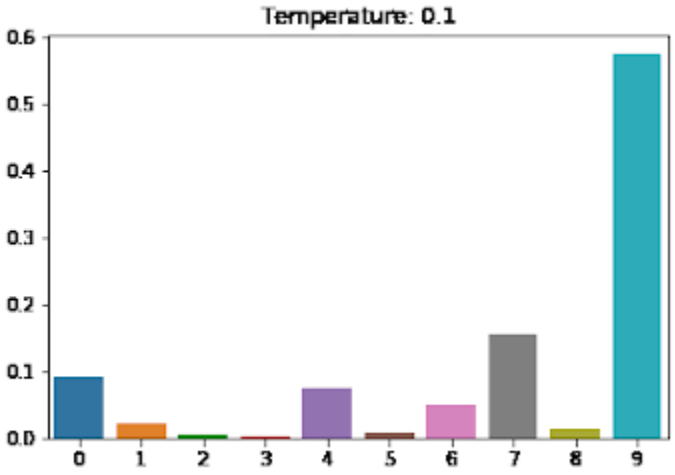
출처: https://medium.com/@harshit158/softmax-temperature-5492e4007f71



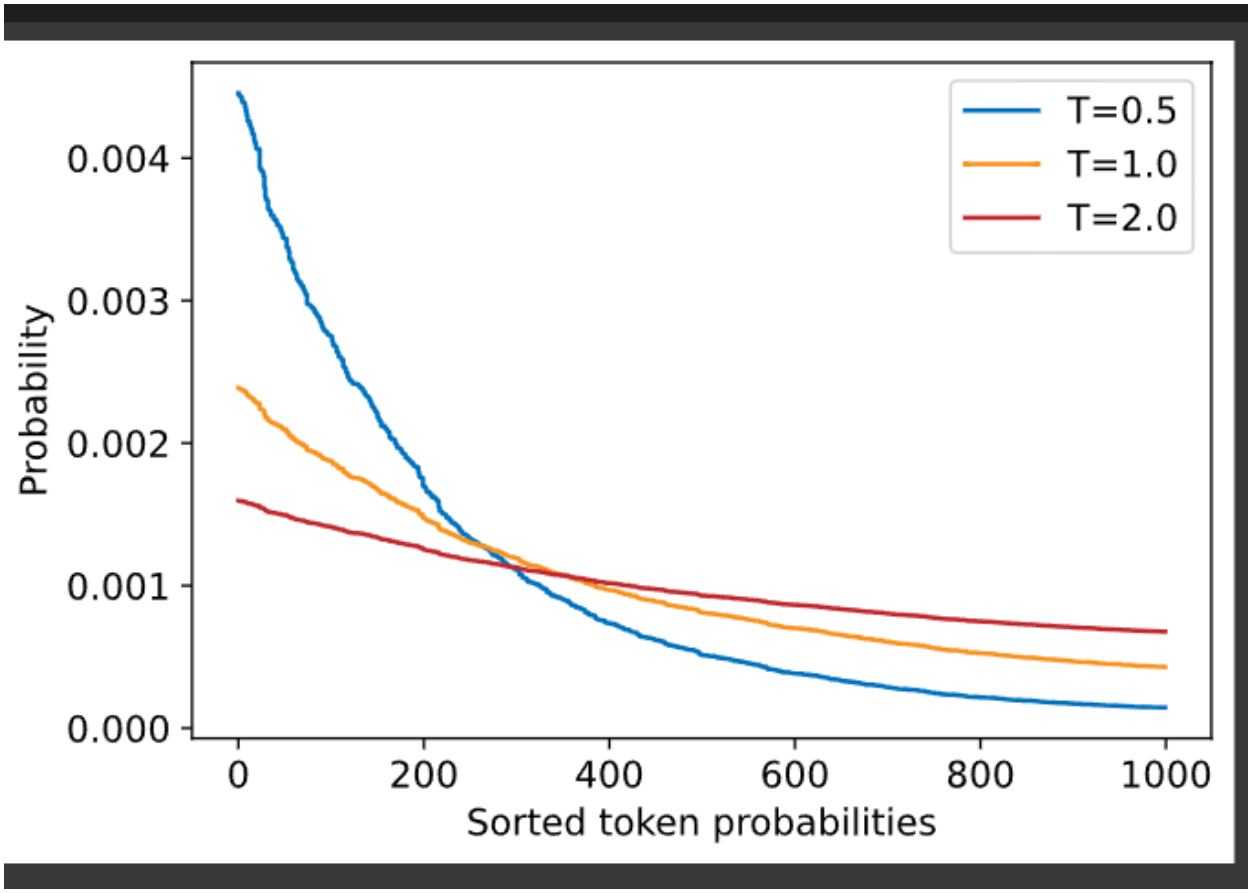
온도를 높이면 자유도가 높아져서 아무거나 random하게 선택될 확률이 높아지고,
온도가 낮으면 반대임.

The effect of this scaling can be visualized in Fig 3:





위의 그래프를 온도별로 내림차순으로 sorting한 그래프

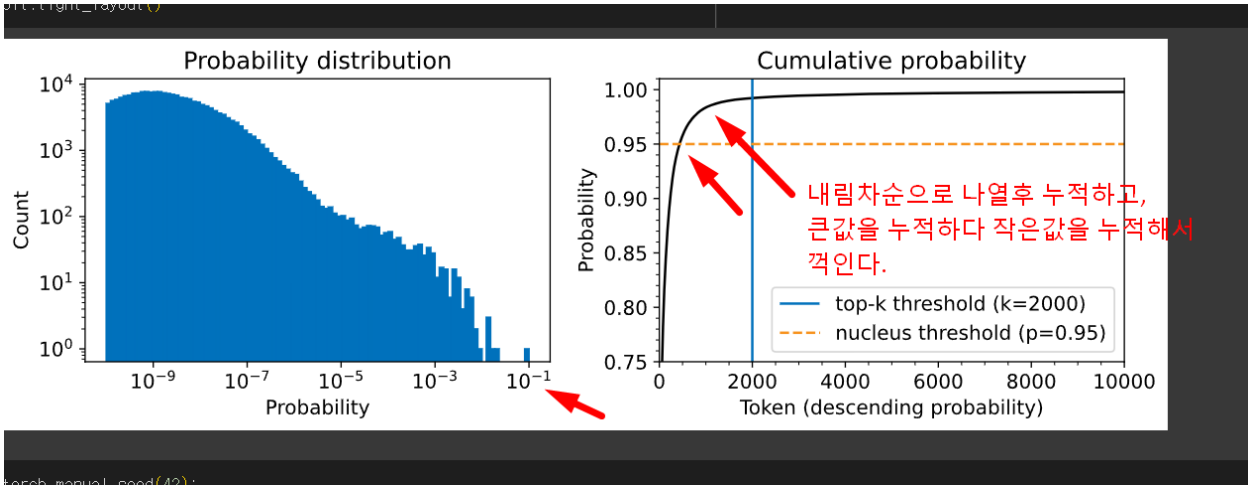


아래그림

내림차순으로 나열후 누적하고,큰값을 누적하다 작은값을 누적해서 꺽인다.

노란색 부분은 ex. top_p=0.90) # 정해서 그 밑에서 갖고옴 -- 누적확률의 임계값을 지정후 그 아래 token만 확률대로 샘플링

top-k sampling은 파란색 세로선처럼 확률이 높은 k개에서 sampling



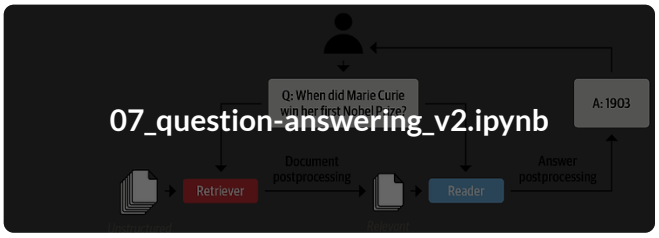
결론

여러가지 decoding 혼합해서 써야 정확도 높아짐

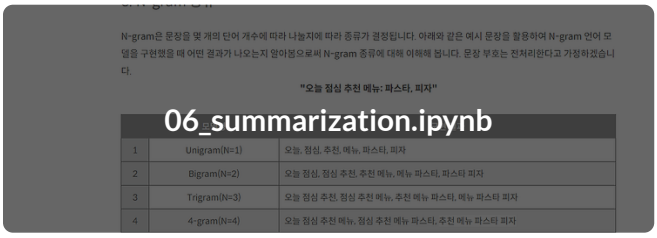
'transformer' 카테고리의 다른 글

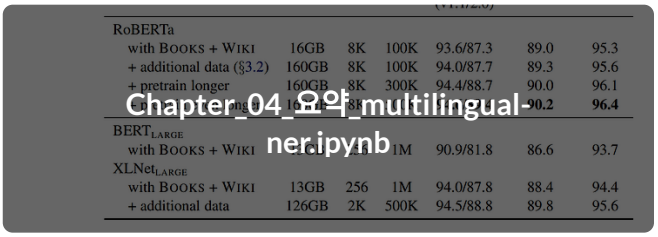
07_question-answering_v2.ipynb (0)	2024.08.16
06_summarization.ipynb (0)	2024.08.07
Chapter_04_요약_multilingual-ner.ipynb (0)	2024.07.26
Chapter_03_요약_transformer 파해치기 https://nlpinkorean.github.io/illustrated-transform... (0)	2024.07.12
F1 score (0)	2024.07.12

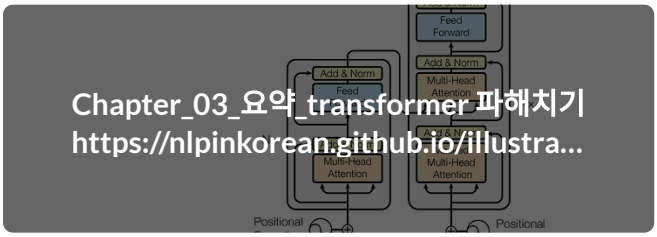
관련글 관련글 더보기



07_question-answering_v2.ipynb


06_summarization.ipynb


Chapter_04_요약_multilingual-ner.ipynb



Chapter_03_요약_transformer 파해치기 https://nlpinkorean.github.io/illustra...

자연어(NLP)

네이쳐2024 님의 블로그입니다.

구독하기 +

댓글 0

이름

비밀번호

내용을 입력하세요.

등록