2. 언어 모델

<블록 시작>

어느 한가로운 일요일에 윤우아빠와 윤우는 누가 누가 더 속담을 많이 맞추는지 놀이를 했다.

엄마: 가는 날이 OOOO

윤우아빠: 장날이다

윤우: 대박이다

엄마: 등잔 밑이 OOO

윤우아빠: 어둡다

윤우: 뜨겁다

엄마: 백지장도 맛들면 OO

윤우아빠: 낫다

윤우: 백지장이 뭐야?

그렇게 속담놀이는 윤우아빠의 완승으로 끝났다고 한다.

<블록 끝>

위의 이야기에서 윤우아빠의 완승으로 끝난 이유는 윤우아빠가 윤우보다 훨씬 오래 살았고, 그래서 윤우보다 훨씬 더 많은 속담을 반복적으로 들었기 때문이다. 윤우아빠가 살아오면서 "등잔 밑이"라는 말 다음에 들었던 것은 대부분 "어둡다"였다. 반면에 아직 몇 년 살지 않은 윤우는 "등잔 밑이" 다음에 무슨 말이 오는지 들어본 적도 없다. 그래서 상상의 나래를 펼쳐서 "뜨겁다"라는 답을 한 것이다. "백지장도 맛들면"도 마찬가지이다. 윤우아빠가 살면서 다음 단어로 가장 많이 들은 것은 "낫다"이다. 그런데 윤우는 백지장이라는 단어를 처음 들어본다. 그래서 백지장이 뭐냐고 되려 질문을 한 것이다.

2.1. 언어 모델은 확률 게임?

전통적으로 언어 모델이란 단어들을 확률적으로 나타낸 것이다. 확률적으로 어떻게 표현할지는 언어 모델마다 다르다. Unigram 방식의 언어 모델과 N-Gram 방식의 언어 모델의 경우, 단어를 표현하는 기본적인 가정을 다르게 한다. Unigram의 경우 나열된 단어의 확률을 각 단어의 확률을 곱한 것으로 나타낸다.

<블록 시작>

내 이름은 홍길동이다.

P(내) = 0.2

P(이름은) = 0.4

P(홍길동이다) = 0.1

P(내, 이름은, 홍길동이다)

= P(내) x P(이름은) x P(홍길동이다)

= 0.2 x 0.4 x 0.1

<블록 끝>

##각주1 P(w)는 w라는 단어가 나올 확률을 뜻한다.

이를 수식으로 표현하면 [수식1]과 같다.

<수식 시작>

Unigram 수식

수식1

<수식 끝>

[수식1]에서는 단어의 순서가 고려되지 않았다. "내 이름은 홍길동이다"와 "이름은 내 홍길동이다"가 같은 확률을 갖게 된다. 그래서 Unigram 방식의 언어 모델은 거의 사용되지 않는다. 이 방식을 개선한 방법이 BiGram 기반의 언어 모델이다.

<블록 시작>

내 이름은 홍길동이다.

P(내) = 0.2

P(이름은|내) = 0.1

P(홍길동이다|이름은) = 0.01

P(내, 이름은, 홍길동이다)

= P(내) x P(이름은|내) x P(홍길동이다|이름은)

= 0.2 x 0.1 x 0.01

<블록 끝>

##각주2 P(w1|w2)는 w2라는 단어가 나왔을 때 w1이 나올 확률을 표시하며 조건부 확률이라고 한다.

이를 수식으로 표현하면 [수식2]과 같다.

<수식 시작>

Bigram 수식

수식2

<수식 끝>

[수식2]는 단어의 확률은 이전 단어에 의해 결정된다는 가정을 하고 있다. Bigram의 경우 P(내, 이름은, 홍길동이다) = P(내) x P(이름은|내) x P(홍길동이다|이름은)라는 가정하에 만들어진 언어 모델인 것이다. 이렇게 단어의 순서를 고려한 언어 모델의 경우 "내 이름은 홍길동이다"와 "이름은 내 홍길동이다"가 서로 다른 확률을 나타낼 것이다. 정상적인 말뭉치로 학습했다면 당연히 "내 이름은 홍길동이다"의 확률이 훨씬 높게 나올 것이다. [수식2]에서는 단어의 확률을 두 단어의 순서 의해 결정된다고 표현했는데, 단어 N개의 순서에 의해 결정된다고 하면 N-gram 방식의 언어 모델이 된다. N-gram의 수식은 [수식3]과 같다.

<수식 시작>

N-gram 수식

수식3

<수식 끝>

<팁 시작>

수식이 나왔다고 좌절할 필요가 없다. 수식은 머릿 속으로 이해할 수 있는 것을 기호로 표현한 것일 뿐이다. 수식에 거부감이 있다면 "내 이름은 홍길동이다"라는 예시를 통해 원리를 충분히 이해한 후 그것을 기호 표현으로 일반화하려면 어떻게 표현하면 되는지를 생각해보자. 그 다음에 수식을 보면 조금 이해가 수월해질 것이다.

<팁 끝>

자연어 처리에서 언어 모델이 왜 필요할까? 최대한 현실적으로 언어를 만들어내기 위함이다. 사람들이 보통 "나"라는 단어 다음에 "는"라는 단어를 많이 쓰더라. "는"라는 단어 다음에 "입니다"라는 단어는 나올 확률이 극히 드물더라. 이런 것을 확률적으로 컴퓨터에게 학습시켜야 컴퓨터가 최대한 인간과 유사한 언어를 구사하게 되기 때문이다.

역사적으로 보면 언어 모델에 대한 연구는 대부분 확률 기반이었다. 그러다가 딥러닝 기반의 언어 모델 연구가 2015년 이후에 활발해졌다. 확률 기반의 언어 모델의 경우 언어 모델의 결과가 보통 확률에서 끝난다. "내 이름은 홍길동이다"라는 문장의 확률 또는 그 다음에 나올 단어의 확률로 끝나게 된다. 반면에 딥러닝 기반의 언어 모델은 하나의 문장을 어떤 벡터로 표현하는 결과를 보인다. 그 과정에서 확률적인 접근이 들어가는 경우도 있고 그렇지 않은 경우도 있다. 이 책에서는 딥러닝 기반의 언어 모델에 집중을 하고자 한다. 딥러닝 기반의 언어 모델은 이번 장에서도 다루겠지만 본격적인 내용은 5장에서 다루려고 한다.

다음 절에서는 N-gram 언어모델을 구현해보자.

2.2. N-gram 언어 모델 만들기

이번 절에서는 N-gram 언어 모델을 만들어려고 한다. 언어 모델을 만든다는 것을 그렇게 어렵게 생각하지 말자. N-gram 언어 모델의 경우에는 결국 확률 기반의 언어 모델이기 때문에 기본적으로 특정 단어의 개수를 전체 단어의 개수로 나누는 기본 공식을 가지고 구현하면 된다.

2.2.1. 텍스트 전처리 함수

가장 먼저 구현해야 하는 것은 텍스트 전처리 함수이다. 텍스트 전처리 함수에서는 아래의 작업을 수행한다.

<블록 시작>

STEP1: 소문자 치환하기

STEP2: BOS, EOS 추가하기

STEP3: 토큰화하기

STEP4: 한번 출현한 단어 UNK으로 치환하기

<블록 끝>

우선 소문자 치환하기이다. 소문자 치환은 해도 되고 안해도 된다. 각종 문장에 대한 전처리를 이 단계에서 하면 된다. 예를 들어 특수문자를 제거하는 등의 케릭터 단위로 수행되는 함수들은 이 단계에서 수행하면 된다. 이 책에서는 [코드1]과 같이 간단하게 소문자 치환하는 것만 대표적으로 구현하려고 한다.

<코드 시작>

코드1

# STEP1: 소문자 치환하기

sentences = list(map(str.lower, sentences))

<코드 끝>

그 다음으로 BOS와 EOS를 추가하는 함수를 구현해보자. BOS는 Beginning of sentence의 약자이며 <s>로 나타내고, EOS는 End of sentence의 약자이고 </s>로 나타낸다. BOS와 EOS는 자연어 처리를 할 때 필요한 예약어이다. <s>를 문장 앞에 놓아서 문장의 시작을 알리고 </s>를 문장 끝에 놓아서 문장의 끝을 알리는 역할을 한다. 보통은 각 문장당 양쪽에 하나씩 추가하지만, 여기에서는 N-gram 모델을 만들기 때문에 BOS는 N-1개 추가할 것이다. 왜 EOS는 항상 1개 추가하면서 BOS는 N-1개를 추가할까? 시작하는 단어의 확률을 구하기 위해서이다. 가령 N=3일때, I로 시작하는 문장에서 제일 처음에 I라는 단어에 대한 확률을 구할 때는 P(I|<s><s>)와 같이 구해야 하기 때문이다. [코드2]를 참고하라.

<코드 시작>

코드2

# STEP2: BOS, EOS 추가하기

BOS = '<s>'

EOS = '</s>'

BOSs = ' '.join([BOS]\*(n-1) if n > 1 else [BOS])

sentences = [' '.join([BOSs, s, EOS]) for s in sentences]

<코드 끝>

이제 각 문장을 토큰화해서 하나의 큰 리스트로 만들자. 토큰화는 띄어쓰기(' ')를 기준으로 토큰화했다. 토큰화된 리스트의 리스트 형태를 하나의 큰 리스트로 합치는 과정은 [코드3]과 같이 reduce를 이용해 구현했다.

<코드 시작>

코드3

# STEP3: 토큰화하기

from functools import reduce

sentences = list(map(lambda s: s.split(), sentences))

tokens = list(reduce(lambda a, b: a+b, sentences))

<코드 끝>

마지막으로 [코드4]에서 단어가 출현한 빈도가 1인 경우는 <unk>으로 치환하는 것을 구현했다. 빈도가 너무 낮은 단어는 무시해서 모르는 단어로 취급하는 것이다. 지나친 디테일을 피하기 위한 과정으로 필수적인 과정은 아니다. 여기에서는 예시로 빈도가 1인 경우에 <unk>으로 치환하지만 말뭉치 사이즈에 따라서 이 숫자를 조금 더 키워도 된다. <unk>은 BOS나 EOS와 같이 예약어이며 출현하지 않은 단어를 나타내기 위해 사용된다.

<코드 시작>

코드4

# STEP4: 한번 출현한 단어 UNK으로 치환하기

import nltk

UNK = '<unk>'

freq = nltk.FreqDist(tokens)

tokens = [t if freq[t] > 1 else UNK for t in tokens]

<코드 끝>

STEP1부터 STEP4까지 하나의 preprocess 함수안에 넣으면 [코드5]와 같다.

<코드 시작>

코드5

import nltk

from functools import reduce

def preprocess(sentences, n):

'''Split list of sentences into words and make a list of words

Args:

sentences (list of str): a list of sentences

n (int): variable N for N-gram model. How many words you want to see before a target word.

Returns:

A list of tokens which were tokenized from each sentence

'''

BOS = '<s>'

EOS = '</s>'

UNK = '<unk>'

# STEP1: 소문자 치환하기

sentences = list(map(str.lower, sentences))

# STEP2: BOS, EOS 추가하기

BOSs = ' '.join([BOS]\*(n-1) if n > 1 else [BOS])

sentences = [' '.join([BOSs, s, EOS]) for s in sentences]

# STEP3: 토큰화하기

sentences = list(map(lambda s: s.split(), sentences))

tokens = list(reduce(lambda a, b: a+b, sentences))

# STEP4: 한번 출현한 단어 UNK으로 치환하기

freq = nltk.FreqDist(tokens)

tokens = [t if freq[t] > 1 else UNK for t in tokens]

return tokens

<코드 끝>

2.2.2. 제로 카운트 해결하기

N-gram 언어 모델은 단어의 빈도수를 이용해서 각 단어에 대한 확률을 구한다. 그러면 한번도 나타나지 않은 단어에 대해서는 어떻게 확률을 구할까? Laplace Smoothing을 이용해서 해결해줄 수 있다.

<수식 시작>

laplace 수식

수식4

<수식 끝>

[수식4]와 같이 모든 단어가 최소 한번 이상 나타났다고 가정하는 것이다. 분자에 무조건 1을 더해준다. 그런데 분자에만 1을 더해주면 확률의 합이 1이 되지 않으므로 분모에 전체 단어 수를 더해준다. 전체 단어 수만큼 더해주는 이유는 분자에 1을 더해도 확률의 합이 1이 되게 하기 위해서이다. Laplace smoothing을 하면 어떻게 확률을 구할 수 있는지 아래의 예시를 통해 알아보자.

<블록 시작>

she sells sea-shells by the sea-shore.

the shells she sells are sea-shells, I'm sure.

for if she sells sea-shells by the sea-shore

then i'm sure she sells sea-shore shells.

출처: https://en.wikipedia.org/wiki/Tongue-twister

<블록 끝>

위의 예시를 n=2로 설정해서 preprocess() 함수를 사용하면 [코드6]과 같은 결과를 얻을 수 있다.

<코드 시작>

>>> import nltk

>>> from language\_model import preprocess

>>> text = '''she sells sea-shells by the sea-shore. the shells she sells are sea-shells, I'm sure. for if she sells sea-shells by the sea-shore then i'm sure she sells sea-shore shells.'''

>>> sentence\_list = nltk.tokenize.sent\_tokenize(text)

>>> words = preprocess(sentence\_list, n=2)

>>> bigram = nltk.ngrams(words, n=2)

>>> vocab = nltk.FreqDist(bigram)

>>> for k, c in vocab.items():

... a, b = k

... print('{},{}: {}'.format(a, b, c))

...

<s>,she: 1

she,sells: 4

sells,sea-shells: 2

sea-shells,by: 2

by,the: 2

the,<unk>: 2

<unk>,</s>: 3

</s>,<s>: 2

<s>,the: 1

<unk>,she: 3

sells,<unk>: 1

<unk>,<unk>: 2

<unk>,i'm: 2

i'm,<unk>: 2

<s>,<unk>: 1

the,sea-shore: 1

sea-shore,<unk>: 2

sells,sea-shore: 1

<코드 끝>

[코드6]의 `text`를 bigram으로 vocab을 만들고 그것을 표로 나타내면 아래와 같다.

<표 시작>

표1

표2

표3

language\_model.py에서 draw\_table1()

language\_model.py에서 draw\_table2()

language\_model.py에서 draw\_table3()

<표 끝>

[표1]에서 the는 총 3번 나왔다. the에 대한 bigram 카운트를 보면 by,the가 2번, <s>,the가 1번 나와서 총 3번 나왔다. 그 외의 단어 she, sells 등등의 단어 다음에는 the가 나온 적이 없다. 그럼어도 불구하고 [표3]와 같이 적어도 한번은 나왔다고 카운트 해주는 것이다. 그 다음에 [수식4]를 이용해서 각 bigram 카운트에 대한 확률을 구하면 [표4]과 같다.

<표 시작>

표4

language\_model.py에서 draw\_table4()

<표 끝>

계산법을 더 자세하게 설명해보자. 가령 P(she,sells|sells)의 확률은 0.1111이다. 이는 (4+1)/(4+41)이며 she,sells의 빈도가 분자의 4이고 sells의 빈도가 분모의 4이고 총 단어의 수는 41이기 때문이다.

Laplace smoothing은 1을 더해주는 방식의 smoothing 기법이기 때문에 Add-one smoothing이라고도 부른다. 여기에서 1을 더하는 대신 어떤 수 k를 더한다고 한다면 Add-k smoothing이 된다. Add-k smoothing의 공식은 아래와 같다.

<수식 시작>

수식5

<수식 끝>

지금까지 Laplace smoothing 기법을 이용해서 제로 카운트 문제를 해결하는 법을 알아봤다. 이 방법을 이용하면 한번도 나타나지 않은 단어에 대해서 확률을 구하지 못하는 경우는 없어진다.

제로 카운트를 해결하는 또 다른 방법으로 Backoff와 Interpolation이 있다. 예를 들어서 아래와 같이 3-gram 언어 모델을 이용해서 문장의 확률을 구해보자.

<블록 시작>

<s> I love you because you love me. </s>

P(love|<s>,I) = 0.1

P(you|I,love) = 0.01

P(because|love,you) = 0.02

P(you|you,because) = 0.05

P(love|because,you) = 0.03

P(me|you,love) = 0.09

P(</s>|love,me) = ?

P(<s> I love you because you love me. </s>)

= 0.1 x 0.01 x 0.02 x 0.05 x 0.03 x 0.09 x ?

<블록 끝>

love me 다음에 문장이 끝난(</s>) 경우가 없다고 가정해보자. 그런데 만일 2-gram으로 언어 모델을 계산했을때 P(</s>|me)의 확률이 있다면 그것을 대신해서 써도 되지 않을까? 만일 그것도 없다면 P(</s>)와 같이 Unigram 모델의 형태로 대신해서 쓰는게 더 좋지 않을까? 이 개념이 Backoff이다. N-gram 언어 모델을 이용해서 각 단어 조합의 경우의 수 확률을 구할 때 만일 그 N-gram이 없다면 N-1 gram에서 확률을 찾아 사용하고, (N-1)-gram에도 없다면 (N-2)-gram에서 찾아서 사용하는 방식을 사용하는 것이다.

<수식 시작>

수식6

Backoff 수식

<수식 끝>

Interpolation은 N-gram, (N-1)-gram, (N-2)-gram, ...을 일정 비율로 곱한 것이다.

<수식 시작>

수식6

Interpolation 수식

<수식 끝>

이 외에도 디스카운팅이나 Kneser-Ney Smoothing과 같이 N-gram 언어 모델의 성능을 높이는데 도움이 되는 기법들이 있으나 이 책에서 더 자세하게 설명하지는 않으려고 한다.

2.2.3. N-gram 모델 학습하기

전처리된 토큰들을 이용해서 N-gram 모델을 만들어보자. N은 1 이상의 자연수이고 N=1일 경우와 그렇지 않은 경우로 구현방법이 나뉘기 때문에 N=1을 기준으로 분기하여 코드를 구현했다. 그리고 단어의 빈도는 nltk에서 제공하는 nltk.ngrams와 nltk.FreqDist를 이용해서 구현했고 사용 예시는 [코드7]를 참고하면 된다.

<코드 시작>

코드7

>>> import nltk

>>> a = ['a','b','b','b','a','a','a','c']

>>> bigram = nltk.ngrams(a, n=2)

>>> bigram

<generator object ngrams at 0x7fcfa3904888>

>>> vocab = nltk.FreqDist(bigram)

>>> vocab

FreqDist({('b', 'b'): 2, ('a', 'a'): 2, ('a', 'b'): 1, ('b', 'a'): 1, ('a', 'c'): 1})

<코드 끝>

N-gram 모델을 빌드하기 위해 build\_model 함수를 구현해보자. 이 함수는 토큰화된 단어와 N을 파라미터로 갖는다. nltk.ngrams와 nltk.FreqDist를 이용해서 빈도수를 구한 후 N의 값에 따라 분기로 나눠서 구현할 수 있다. [코드8]을 보자.

<코드 시작>

코드8

def build\_model(self, n):

ngrams = nltk.ngrams(self.tokens, n)

nvocab = nltk.FreqDist(ngrams)

if n == 1:

vocab = nltk.FreqDist(self.tokens)

vocab\_size = len(nvocab)

return {v: c/vocab\_size for v, c in vocab.items()}

else:

mgrams = nltk.ngrams(self.tokens, n-1)

mvocab = nltk.FreqDist(mgrams)

def ngram\_prob(ngram, ncount):

mgram = ngram[:-1]

mcount = mvocab[mgram]

return ncount / mcount

return {v: ngram\_prob(v, c) for v, c in nvocab.items()}

<코드 끝>

n=1일 경우는 간단한 Unigram이기 때문에 추가적인 설명은 생략하려고 한다. n=1이 아닌 경우(else 문)을 보면 n-1을 이용해서도 토큰의 빈도수를 구하는 것을 볼 수 있다.(mgrams와 mvocab) 이는 N-gram 모델의 확률을 구하기 위해서 필요하다. 가령 n=3일 경우에 대해서 빈도를 구하려면 분모 부분에는 n=2일 경우에 대한 빈도가 필요하다.

2.2.4. N-gram 언어 모델의 한계

N-gram 언어 모델의 경우 기본적인 한계를 가지고 있다. 문맥의 부재이다. 확률을 구하고자 하는 단어 바로 앞에 나오는 N-1개의 단어 빈도를 이용했을 뿐이다. 문맥을 특별히 고려하지 않았다. 단순히 특정 단어들이 학습 데이터셋에서 연속적으로 몇 번 나왔는지를 기반으로 학습된 모델이다. 2.2.2절에서 설명한 제로 카운트 해결하는 방법인 Laplace smoothing, Backoff, Interpolation 등의 기법이 나온 이유 역시 이러한 문맥의 부재를 조금이나마 덜기 위해서다. Backoff로 예를 든다면, N-gram 확률이 없을 경우 (N-1)-gram의 확률로 대체하는데 이 방법이 문맥의 부재를 해결하는 근본적인 해결책은 될 수 없다.

이런 문제를 해결하기 위한 접근 방법이 딥러닝 기반의 언어 모델이다. 딥러닝 기반의 언어 모델로 가장 오래전부터 연구돼 왔던 모델은 RNN 기반의 모델이다. 다음 절에서는 RNN 기반의 언어 모델에 대해서 자세하게 알아보도록 하자.

2.3. 딥러닝 기반의 언어 모델

딥러닝은 1960년대부터 연구됐던 학문이다. 하지만 당시에는 하드웨어 자원이 충분치 않아서 연구로만 끝났다가 2015년 이후 하드웨어 자원이 충분해진 이후 급속도로 발전했다. 확률 기반의 언어 모델과 딥러닝 기반의 언어모델의 가장 큰 차이점은 무엇일까? 확률 기반은 시퀀스의 조합을 고정된 확률로 나타낸다. 반면에 딥러닝은 표현을 컨텍스트를 표현하는 방법을 학습하는 학문이다. 여기에서 말하는 컨텍스트는 다루는 데이터에 따라 다르다. 이미지의 경우 컨텍스트는 픽셀의 RGB 값이 될 수 있을 것이고, 자연어의 경우에는 우리가 크면서 자연스럽게 배우는 문법, 문맥, 어감 등이 될 것이다. 딥러닝 기반의 언어 모델은 문맥을 표현하는 방법을 학습하고, 이를 쉽게 말해서 문맥을 학습한다고 한다.

2.3.1. 입력 데이터의 벡터화

딥러닝을 인터넷에 검색해서 이미지를 보면 아래와 같은 그림을 가장 많이 볼 수 있을 것이다.

<그림 시작>

동그라미 여러개 여러 계층 그림 가장 기본적인거

[1,2] x [[2,1,0,3],[1,4,3,0]] = [4,9,6,3]

그림1

<그림 끝>

[그림1]은 딥러닝에서 가장 기본적인 연산인 행렬 곱을 그림으로 표현한 것이다. 행렬 곱을 통해서 딥러닝 모델의 입력 값을 어떤 다른 벡터로 변환해버릴 수 있다.

[그림1]에서 [1,2]에 행렬 곱 한번만 했을뿐인데 [4,9,6,3]이라는 벡터로 변해버렸다. 해석하자면 이 벡터는 입력 데이터인 [1,2]를 표현하는 벡터인 것이다. 입력 데이터를 다른 벡터로 표현하는 방법은 행렬 곱 이외에도 여러가지가 있다. 이미지 데이터를 다룰 경우에는 CNN 구조를 많이 사용하고 텍스트 데이터를 다루는 경우에는 RNN 구조를 사용한다. RNN 구조는 순서를 갖는 데이터를 다룰 때 많이 사용하는 구조이다. 딥러닝에서는 전통적으로 텍스트의 경우 단어의 순서가 중요한 학습 포인트가 되기 때문에 RNN 구조를 많이 사용해왔다. RNN 구조의 경우 short-term memory 문제가 있다. 즉 긴 단어를 기억하지 못하는 문제이다. 그렇기 때문에 이 장에서는 이 문제를 보완한 GRU 구조를 이용해서 언어 모델을 만들어보려고 한다. 실제로 구현을 하기 전에 다음 절에서 GRU 구조에 대해서 설명을 하고 왜 GRU를 텍스트 데이터를 다루는 경우에 사용해왔는지 그 이유를 함께 이해해보자.

2.3.2. GRU 입출력 구조 이해해보기

GRU 구조는 하나의 입력 벡터(x\_i)와 은닉 벡터(h\_i-1)를 입력으로 취하고 은닉 벡터(h\_i)와 출력 벡터(O\_i)를 만들어 낸다.

<그림 시작>

GRU 하나 입출력 셀 그림 자세하게

그림2

<그림 끝>

[그림2]의 입력 벡터가 단어 하나이다. 자연어 처리에서는 여러 개의 단어를 처리해야 하므로 [그림2]과 같은 구조를 [그림3]와 같이 여러번 반복해 보자.

<그림 시작>

GRU 여러개 입출력 셀 그림 자세하게

I love you

그림3

<그림 끝>

GRU는 [그림3]와 같이 그림으로 표현하면 더욱 쉽게 이해할 수 있다. [그림3]에서 원래의 입력 텍스트는 "I love you"였다. 이것을 각각 I=x\_0=[1,0,0], love=x\_1=[0,1,0], you=x\_2=[0,0,1]과 같이 one-hot 벡터로 표현해서 하나씩 RNN 셀에 넣었다. x\_0을 넣을 때 h\_0=[0,0,0,0]을 같이 넣었고 그 결과로 h\_1=[1,1,1,1]을 얻었다. h\_1을 다시 x\_1과 함께 GRU에 넣으니 h\_2가 출력됐고, 같은 원리로 h\_3까지 출력된 것을 이해할 수 있다. h\_3이 "I love you"를 표현한 벡터가 되는 것이다. 만일 입력 문장이 "I love him" 이었으면 h\_3는 다른 값을 갖게 될 것이다. 꼭 GRU가 아니여도 RNN 기반의 구조는 모두 이와 같이 입력 데이터를 순서대로 넣는 구조를 취하기 때문에 RNN 구조의 딥러닝 모델이 텍스트 처리에 자주 사용돼 왔다.

GRU 구조 하나가 위와 같이 입력과 출력을 가질 수 있는 이유는 GRU가 내부적으로 아래와 같은 복잡한 구조를 하고 있기 때문이다.

<그림 시작>

복잡한 GRU 내부 구조

그림4

<그림 끝>

이 책에서는 GRU 내부 구조의 계산 과정보다는 전체적인 입출력의 벡터 사이즈를 더 자세하게 설명하려고 한다. GRU 내부의 계산 과정을 더 자세하게 알아보고 싶다면 아래의 링크를 참고하라.

(## GRU 내부 계산과정 레퍼런스 추가)

GRU의 입출력을 간단하게 볼 수 있는 샘플 코드를 보자.

<코드 시작>

코드9

>>> import torch

>>> from torch import nn

>>>

>>> num\_layers = 2

>>> input\_size = 10

>>> hidden\_size = 20

>>> batch\_size = 3

>>> seq\_len = 5

>>>

>>> rnn = nn.GRU(input\_size, hidden\_size, num\_layers, batch\_first=True)

>>> input = torch.randn(batch\_size, seq\_len, input\_size)

>>> h0 = torch.randn(num\_layers, batch\_size, hidden\_size)

>>> output, hn = rnn(input, h0)

>>> output.shape, hn.shape

(torch.Size([3, 5, 20]), torch.Size([2, 3, 20]))

<코드 끝>

[코드9]에서 input\_size는 x를 표현하는 피처의 크기이다. 예를 들어 [그림3]에서 I=x\_0=[1,0,0]과 같이 x\_0을 3차원의 벡터로 표현했다. 이 경우 input\_size는 3이 된다. hidden\_size는 x\_0를 표현하는 사이즈이다. [그림3]에서는 h\_1=[1,1,1,1]과 같이 4차원으로 표현했는데 이 경우에는 hidden\_size는 4가 된다. input\_size와 hidden\_size를 정하면 기본적으로 GRU 계층 하나를 만들 수 있다. 이 GRU 계층을 여러 개 쌓아서 stacked-GRU를 만들 수도 있는데 그 때 몇 개를 쌓는지를 num\_layers를 통해서 결정할 수 있다. batch\_size는 한번에 처리하는 데이터의 개수이며 seq\_len은 입력 데이터의 길이이다. [그림3]의 경우 [I,love,you] 세 단어로 구성돼 있으니 [그림3]의 경우 seq\_len은 3이다.

이번 절에서는 GRU의 입출력 구조에 대해서 알아봤다. 다음 절에서는 GRU를 이용해서 언어 모델을 구현해보자.

2.3.3. GRU 언어 모델 구현하기

이번 절에서는 GRU를 이용해서 간단한 언어 모델을 구현해보려고 한다. 거창하게 들리지만 결국 다음 단어를 잘 예측하는 모델을 만들는 것이다.

GRU 언어 모델에 대한 입력과 출력을 정의해보자. [그림3]를 보면 GRU는 [I,love,you]라는 단어를 순서대로 입력으로 취하고 있고, 각 단계마다 출력(O\_0, O\_1, O\_2)을 하고 있다. 그러면 I가 입력으로 들어갔을 때 love가 출력되고, love가 입력으로 들어갔을 때 you가 출력되도록 학습하면 된다. 여기에 BOS(<s>)와 EOS(</s>)를 추가해서 정리하면 아래와 같이 정리할 수 있다.

<블록 시작>

원문: I love you

BOS,EOS 추가 후 토큰화: [<s>, I, love, you, </s>]

모델 입력: [<s>, I, love, you]

모델 출력: [I, love, you, </s>]

<블록 끝>

<팁 시작>

실무에서 언어 모델을 학습할 때 데이터를 전처리하는 부분에 대해서는 최대한 꼼꼼하게 검토하고 여러 테스트 케이스를 작성해서 검토해야 한다. 실무에서 만드는 언어 모델의 경우 데이터의 양이 굉장히 많을 수 밖에 없고 학습시간도 오래 걸리기 때문에 데이터 전처리 부분에서 최대한 오류가 없어야 하며 코드도 깔끔하게 작성해야 한다. 예를 들어 모델 학습을 1주일째 진행하고 있는데 데이터 전처리 단계에서 어떤 문제가 발생하게 될 경우 시간을 크게 낭비하게 된다.

<팁 끝>

<s>가 입력됐을 때 I가 출력되도록,

I가 입력됐을 때 love가 출력되도록,

love가 입력됐을 때 you가 출력되도록,

you가 입력됐을 때 </s>가 출력되도록 Dataset 클래스 작성하자.

<코드 시작>

코드10

import torch

import nltk

from functools import reduce

from torch.utils.data import Dataset

def preprocess(sentences, add\_special\_tokens=True):

'''Split list of sentences into words and make a list of words

Args:

sentences (list of str): a list of sentences

Returns:

A list of tokens which were tokenized from each sentence

'''

BOS = '<s>'

EOS = '</s>'

UNK = '<unk>'

# STEP1: 소문자 치환하기

sentences = list(map(str.lower, sentences))

# STEP2: BOS, EOS 추가하기

if add\_special\_tokens:

sentences = [' '.join([BOS, s, EOS]) for s in sentences]

# STEP3: 토큰화하기

sentences = list(map(lambda s: s.split(), sentences))

return sentences

class GRULanguageModelDataset(Dataset):

'''

GRU 언어 모델을 위한 Dataset 클래스.

Args:

text: 전체 말뭉치 데이터셋

Returns:

토큰화된 text를 텐서 객체로 변환하는 Dataset 클래스

Example:

>>> text = 'I love you'

>>> dataset = GRULanguageModelDataset(text)

>>> for d in dataset:

... print(d)

... break

...

tensor([1, 4, 5, 6, 2])

'''

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

sentence\_list = nltk.tokenize.sent\_tokenize(text)

tokenized\_sentences = preprocess(sentence\_list)

tokens = list(reduce(lambda a, b: a+b, tokenized\_sentences))

self.vocab = self.make\_vocab(tokens)

self.i2v = {v:k for k, v in self.vocab.items()}

self.indice = list(map(lambda s: self.convert\_tokens\_to\_indice(s), tokenized\_sentences))

def convert\_tokens\_to\_indice(self, sentence):

indice = []

for s in sentence:

try:

indice.append(self.vocab[s])

except KeyError:

indice.append(self.vocab['<unk>'])

return torch.tensor(indice)

def make\_vocab(self, tokens):

vocab = {}

vocab['<pad>'] = 0

vocab['<s>'] = 1

vocab['</s>'] = 2

vocab['<unk>'] = 3

index = 4

for t in tokens:

try:

vocab[t]

continue

except KeyError:

vocab[t] = index

index += 1

return vocab

def \_\_len\_\_(self):

return len(self.indice)

def \_\_getitem\_\_(self, idx):

return self.indice[idx]

<코드 끝>

[코드10]의 GRULanguageModelDataset은 아래와 같이 실행할 수 있다.

<코드 시작>

코드11

>>> from dataset import GRULanguageModelDataset

>>> text = 'she sells sea shells by the sea shore'

>>> dataset = GRULanguageModelDataset(text)

>>> for d in dataset:

... print(d)

... break

...

tensor([ 1, 4, 5, 6, 7, 8, 9, 6, 10, 2])

>>> dataset.vocab

{'<pad>': 0, '<s>': 1, '</s>': 2, '<unk>': 3, 'she': 4, 'sells': 5, 'sea': 6, 'shells': 7, 'by': 8, 'the': 9, 'shore': 10}

<코드 끝>

[코드11]을 보면 GRULanguageModelDataset 데이터셋 클래스에서 text에 <s>와 </s>를 앞뒤로 붙이고 토큰화했다. 그 다음에 dataset.vocab을 이용해서 토큰화된 단어들을 숫자로 바꿔주고 텐서 객체로 만든 것이다.

Pytorch를 이용해서 딥러닝 모델을 만들때 torch.nn.Dataset을 상속받아 이용하면 데이터 전처리하는 과정을 구조적으로 간단하게 구현할 수 있다. torch.nn.Dataset 클래스는 \_\_len\_\_ 함수와 \_\_getitem\_\_ 함수를 오버라이딩해야 한다. \_\_getitem\_\_ 함수의 경우 생성된 dataset 객체를 dataset[0], dataset[1]과 같이 iterable하게 사용할 수 있게끔 하기 위함이고, \_\_len\_\_ 함수의 경우 dataset을 이용해서 만든 DataLoader를 사용할 때 사용되기 때문이다.

그 다음에는 DataLoader를 만들 차례이다. dataset을 이용해서 [코드12]와 같이 만들면 된다.

<코드 시작>

코드12

>>> from torch.utils.data import DataLoader

>>> from torch.nn.utils.rnn import pad\_sequence

>>>

>>> def collate\_fn(batch):

... batch = pad\_sequence(batch, batch\_first=True)

... return batch

...

>>> dataloader = DataLoader(dataset, collate\_fn=collate\_fn, batch\_size=16)

<코드 끝>

Pytorch를 이용해서 딥러닝 모델을 만들때 torch.nn.DataLoader를 이용하면 데이터 Feeding하는(전처리된 데이터를 모델에 입력하는 과정) 과정을 구조적으로 간단하게 구현할 수 있다. 위에서 사용한 collate\_fn은 dataset에서 리턴된 텐서 객체를 batch\_size 개수만큼 모델에 Feeding하기 바로 직전에 호출된다. 위의 collate\_fn 함수의 경우 pad\_sequence라는 함수를 사용하는데 이는 torch에서 제공하는 함수로 입력되는 데이터의 길이를 특정 길이로 패딩해주는 역할을 한다. 예를 들어서 dataset[0]의 길이가 10이고 dataset[1]의 길이가 15일 경우, dataset[0]의 길이를 15로 맞춰주는 역할을 한다. 길이를 맞춰주지 않고 모델에 Feeding을 하면 입력 텐서의 길이가 다르다는 오류가 발생하게 된다.

dataloader는 iterable하게 [코드13]과 같이 (batch\_size, max\_length) shape의 텐서를 지속적으로 Feeding한다.

<코드 시작>

코드13

>>> for d in dataloader:

... print(d)

... print(d.shape)

... break

...

tensor([[ 1, 4, 5, 6, 7, 8, 9, 6, 10, 2]])

torch.Size([1, 10])

<코드 끝>

위의 dataloader가 batch\_size=16이 아닌 batch\_size=1을 Feeding한 이유는 dataset을 만들때 설정한 text가 너무 짧기 때문이다. text의 길이를 길게 늘리면 batch\_size=16인 데이터를 Feeding 할 수 있다.

데이터 전처리와 Feeding을 구현했으니 이제 모델링을 해보자. [코드14]와 같이 간단하게 만들 수 있다.

<코드 시작>

코드14

from torch import nn

class GRULanguageModel(nn.Module):

def \_\_init\_\_(self, hidden\_size=30, output\_size=10):

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

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

self.gru = nn.GRU(hidden\_size, hidden\_size, batch\_first=True)

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

self.out = nn.Linear(hidden\_size, output\_size)

def forward(self, inputs, hidden):

'''

Input Parameters

- inputs: (B,1)

- hidden: (1,B,H)

Output returns

- output: (B,1,O)

- hidden: (1,B,H)

Example

>>> import torch

>>> from dataset import GRULanguageModelDataset

>>> from run\_gru import GRULanguageModel, text

>>> dataset = GRULanguageModelDataset(text)

>>> hidden\_size = 30

>>> output\_size = len(dataset.vocab)

>>> hidden = torch.zeros((1, 1, hidden\_size))

>>> inputs = dataset[0].unsqueeze(0)

>>> model = GRULanguageModel(hidden\_size=hidden\_size, output\_size=output\_size)

>>> out = model(inputs, hidden)

>>> hidden = torch.zeros((1, 1, hidden\_size))

>>> inputs = dataset[0].unsqueeze(0)

>>> out = model(inputs, hidden)

>>> out[0].shape, out[1].shape

(torch.Size([1, 10, 21]), torch.Size([1, 1, 30]))

'''

embedded = self.embedding(inputs)

output, hidden = self.gru(embedded, hidden)

output = self.softmax(self.out(output))

return output, hidden

<코드 끝>

위의 코드에서 self.embedding은 (batch\_size, max\_length) 형태로 들어오는 텐서를 hidden\_size로 임베딩하기 위한 레이어이다. hidden\_size로 임베딩하는 이유는 하나의 숫자를 hidden\_size 사이즈의 벡터로 임베딩해서 피처를 표현할 수 있는 차원을 넓혀준 것으로 이해하면 된다. self.gru 레이어에서는 3차원의 텐서만을 입력으로 받기 때문에 (batch\_size, max\_length, 1)과 같이 차원만을 맞춰줘도 상관 없겠지만 그렇게 했을 경우 피처를 표현할 수 있는 경우의 수가 크게 줄어들기 때문에 효율적인 학습을 하지 못한다. hidden\_size=1과 hidden\_size=30으로 각각 모델링을 한 후 학습된 모델로 문장을 생성하면 다음과 같이 큰 차이를 나타낸다. hidden\_size 외의 다른 조건은 동일하게 학습했다.

<블록 시작>

# hidden\_size = 1로 학습한 경우,

generated\_sentence: <s> sea sea sea sea sea sea sea sea <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad>

# hidden\_size = 30로 학습한 경우,

generated\_sentence: <s> for if she sells sea shells by the sea shore then i'm sure she sells sea shore shells.

<블록 끝>

GRULanguageModel에 대한 선언 등은 GRULanguageModel의 forward함수에 Example을 참고하면 된다.

데이터 전처리와 Feeding 그리고 모델링까지 살펴봤다. 이제 실제로 모델을 학습하는 [코드15]를 보자.

<코드 시작>

코드15

import torch

import numpy as np

def train(inputs, labels, model, criterion, optimizer, max\_grad\_norm=None):

'''

Input Parameters

- inputs: (B,M)

- labels: (B,M)

Output returns

- loss: calculated loss for one batch tensor

Example

>>> from torch import nn, optim

>>> from dataset import GRULanguageModelDataset

>>> from run\_gru import GRULanguageModel, text

>>> hidden\_size = 30

>>> dataset = GRULanguageModelDataset(text)

>>> output\_size = len(dataset.vocab)

>>> model = GRULanguageModel(hidden\_size=hidden\_size, output\_size=output\_size)

>>> criterion = nn.NLLLoss()

>>> optimizer = optim.SGD(model.parameters(), lr=0.005)

>>> inputs = dataset[0][:-1].unsqueeze(0)

>>> labels = dataset[0][1:].unsqueeze(0)

>>> loss = train(inputs, labels, model, criterion, optimizer, max\_grad\_norm=5.0)

>>> loss

tensor(27.3188, grad\_fn=<AddBackward0>)

'''

batch\_size = inputs.size()[0]

hidden = torch.zeros((1, batch\_size, hidden\_size))

input\_length = inputs.size()[1]

loss = 0

teacher\_forcing = True if np.random.random() < 0.5 else False

lm\_inputs = inputs[:,0].unsqueeze(-1)

for i in range(input\_length):

output, hidden = model(lm\_inputs, hidden)

output = output.squeeze(1)

loss += criterion(output, labels[:,i])

#print('\*\* {} vs {}'.format(lm\_inputs[0,0], labels[0,i]))

if teacher\_forcing:

lm\_inputs = labels[:,i].unsqueeze(-1)

else:

topv, topi = output.topk(1)

lm\_inputs = topi

loss.backward()

if max\_grad\_norm:

torch.nn.utils.clip\_grad\_norm\_(model.parameters(), max\_grad\_norm)

optimizer.step()

return loss

<코드 끝>

학습을 할 떄 teacher forcing 기법을 1/2 확률로 사용하게 된다. teacher forcing 기법은 하나의 시퀀스 데이터에 대해서 학습을 진행할 때 그 시퀀스 데이터에 해당하는 레이블을 하나 하나 가르쳐주는 방식이다. teacher forcing 기법으로 학습할 때는 다음 iteration에서 사용될 lm\_inputs의 값을 labels로부터 받아왔다. 반면에 teacher forcing을 사용하지 않을 경우에는 output 값으로부터 lm\_inputs을 만들었다.

<그림 시작>

그림4

티쳐포싱 그림

<그림 끝>

[그림4]를 보면 teacher forcing 기법을 사용할 경우 ... ## [그림4]로 티처포싱/no티처포싱 설명하는 본문 추가하기

그 다음으로 train 함수에서 설명할 부분은 max\_grad\_norm 부분이다. 저 부분은 gradient clipping을 위한 부분이다. RNN 계열의 모델을 학습할 경우 gradient explosion 등이 발생하기 쉽다. gradient explosion이란 학습을 진행할 때 에러가 급격하게 커져서 모델의 파라미터 업데이트가 지나치게 많이 바뀌는 현상을 의미한다. 보통 학습률(learning rate)을 지정해서 업데이트되는 양을 조절하지만 학습률을 적용했음에도 불구하고 심하게 gradient가 폭발했기 때문에 학습이 제대로 되지 않는 현상이다. 이럴 경우 해결책으로 gradient clipping을 할 수 있는데 gradient clipping이란 gradient가 특정 임계치(threshold) 값을 넘어갈 경우 gradient를 gradient의l2norm 값으로 나눠고 threshold 값을 곱해주는 것이다. [수식5]를 참고하자.

<수식 시작>

if |g| > threshold

gradient = (threshold/l2norm) \* gradient

else

gradient = gradient

수식5

<수식 끝>

train 함수 사용 예시는 [코드15]에 주석으로 설명돼 있는 Example을 참고하면 된다. run\_gru.py를 실행하면 전체 학습 과정을 실행할 수 있다. run\_gru.py를 실행하면 [코드16]과 같이 출력을 한다.

<코드 시작>

코드16

$ python run\_gru.py

0th iteration -> los=58.4429

0th iteration -> los=60.8093

0th iteration -> los=59.7647

1th iteration -> los=57.4985

1th iteration -> los=59.2246

1th iteration -> los=56.9591

2th iteration -> los=56.2083

...

998th iteration -> los=5.0649

999th iteration -> los=19.3040

999th iteration -> los=4.0943

999th iteration -> los=17.3739

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

</s> was generated. therefore finished at 18th iteration

<코드 끝>

2.3.4. GRU 언어 모델로 문장 생성하기

마지막으로 <s>부터 문장을 생성해보자. 문장을 생성하는 generate\_sentence\_from\_bos 함수는 아래와 같이 구현할 수 있다.

<코드 시작>

def generate\_sentence\_from\_bos(model, vocab, bos=1):

'''

Input Parameters

- bos: begin-of-sentence token index. usually 1

Output returns

- generated\_sentence: a sentence generated by the model

Example

>>> import pickle

>>> import torch

>>> from model import GRULanguageModel

>>> from generate import generate\_sentence\_from\_bos

>>> vocab = pickle.load(open('vocab.pickle', 'rb'))

>>> hidden\_size = 30

>>> output\_size = len(vocab)

>>> model = GRULanguageModel(hidden\_size=hidden\_size, output\_size=output\_size)

>>> model.load\_state\_dict(torch.load('gru\_model.bin'))

<All keys matched successfully>

>>> model.eval()

GRULanguageModel(

(embedding): Embedding(21, 30)

(gru): GRU(30, 30, batch\_first=True)

(softmax): LogSoftmax(dim=-1)

(out): Linear(in\_features=30, out\_features=21, bias=True)

)

>>> generated\_text = generate\_sentence\_from\_bos(model, vocab, bos=1)

>>> print('generated sentence: {}'.format(generated\_text))

generated sentence: <s> for if she sells sea shells by the sea shore then i'm sure she sells sea shore shells.

'''

indice = [bos]

hidden = torch.zeros((1, 1, model.hidden\_size))

lm\_inputs = torch.tensor(indice).unsqueeze(-1)

i2v = {v:k for k, v in vocab.items()}

cnt = 0

eos = vocab['</s>']

generated\_sequence = [lm\_inputs[0].data.item()]

while True:

if cnt == 30:

break

output, hidden = model(lm\_inputs, hidden)

output = output.squeeze(1)

topv, topi = output.topk(1)

lm\_inputs = topi

if topi.data.item() == eos:

tokens = list(map(lambda w: i2v[w], generated\_sequence))

generated\_sentence = ' ' .join(tokens)

return generated\_sentence

generated\_sequence.append(topi.data.item())

cnt += 1

print('max iteration reached. therefore finishing forcefully')

tokens = list(map(lambda w: i2v[w], generated\_sequence))

generated\_sentence = ' ' .join(tokens)

return generated\_sentence

<코드 끝>

추론 과정에서는 label이 없기 때문에 모델의 output에서 가장 큰 값인 topi를 찾아서 그 값을 다음 iteration의 입력으로 사용하게 된다. 그러다가 topi가 eos와 같을 경우에는 문장 생성이 종료된 것으로 간주하고 generated\_sentence를 리턴한다. 정상적으로 실행될 경우 아래와 같은 문장을 생성하게 된다.

<블록 시작>

<s> for if she sells sea shells by the sea shore then i'm sure she sells sea shore shells.

<블록 끝>

이번 장에서 언어 모델을 설명했다. 언어 모델을 확률적으로 접근하는 방법과 딥러닝 기반으로 접근하는 방법에 대해서 알아봤다. 다음 장에서는 어탠션에 대해서 알아보려고 한다.