1. 다음 단어는요? 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 맞들면 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.

1.1. 언어 모델은 확률 게임?

언어 모델이란 무엇인지 먼저 알아보자. 언어 모델은 단어들의 확률 분포이다. 우리가 언어를 말하고 쓰고 들을때 사용되는 단어의 나열을 생각해보자. "I am a boy"라는 문장을 보면 I 다음에 나오는 단어로 am이 나올 확률이 is가 나올 확률보다 훨씬 높다. 언어라는 것을 어떤 현상이라고 생각했을 때 I라는 현상 다음에 am이라는 현상이 나올 확률이 is가 나올 확률보다 높고, 이것을 모델링한 것이 언어 모델이다. 그래서 전통적으로 언어 모델은 언어를 확률로 표시했다. 확률로 표시하는 방법은 여러가지 있는데 전통적인 방법으로는 Unigram, Bigram, N-gram등이 있다. Unigram의 경우 [블록1]과 같이 나열된 단어의 확률을 단순하게 곱한 것으로 나타낸다.

<블록 시작>

블록1: Unigram을 이용한 문장 확률

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: Unigram 수식

<수식 끝>

[수식1]에서는 단어의 순서가 고려되지 않았다. "내 이름은 홍길동이다"와 "이름은 내 홍길동이다"가 같은 확률을 갖게 된다. Unigram 모델을 이용하면 "내 이름은 홍길동이다"와 "이름은 내 홍길동이다"가 같은 확률로 나타나는 문장이라는 것이다. 단어들의 순서를 고려하지 않았기 때문이다. 그래서 Unigram 방식의 언어 모델은 거의 사용되지 않는다. 이 방식을 개선한 방법이 [블록2]의 BiGram 기반의 언어 모델이다. BiGram 언어 모델은 조건부 확률을 이용해서 언어의 분포 확률을 표현한 모델이다. 조건부 확률이란 하나의 현상이 이뤄났다는 가정 하에 그 다음 현상이 일어날 확률이다. [블록2]를 보면 "내"가 나왔다는 전제하에 그 다음에 "이름은"이라는 단어가 나올 확률을 P(이름은|내)로 표시하고 이것이 조건부 확률이다.

<블록 시작>

블록2: Bigram을 이용한 문장 확률

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>

"내 이름은 홍길동이다"라는 문장을 Bigram으로 표현하면 [수식2]과 같다.

<수식 시작>

Bigram 수식

수식2: Bigram 수식

<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: N-gram 수식

<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 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 언어 모델을 구현해보자.

1.2. 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.

1.2.1. 텍스트 전처리 함수

가장 먼저 구현해야 하는 것은 텍스트 전처리 함수이다. 이 절에서 사용하는 텍스트 전처리 함수는 [블록3]과 같은 기본적인 작업을 수행하기로 하자. [블록3]의 과정은 설명을 위한 예시일 뿐이다. 실제 업무에서 활용하기 위해서는 더욱 더 발전된 전처리 과정이 필요하다.

<블록 시작>

블록3: 문장 전처리 과정

STEP1: 소문자 치환하기

STEP2: BOS, EOS 추가하기

STEP3: 토큰화하기

STEP4: 한번 출현한 단어 UNK으로 치환하기

<블록 끝>

우선 소문자 치환하기이다. 각종 문장에 대한 전처리를 이 단계에서 하면 된다. 예를 들어 특수문자를 제거하는 등의 캐릭터 단위로 수행되는 함수들은 이 단계에서 수행하면 된다. 이 절에서는 [코드1]과 같이 간단하게 소문자 치환하는 것만 대표적으로 구현하려고 한다. 이 책의 레포지토리에 있는 chapter1/ngram\_lm/language\_model.ipynb를 참고하라.

<코드 시작>

코드1: 소문자 치환하기

# chapter1/ngram\_lm/ngram\_language\_model.ipynb 참고

# STEP1: 소문자 치환하기

>>> sentences = ['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."]

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

>>> sentences

['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."]

<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 한 개 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: BOS/EOS 추가하기

# chapter1/ngram\_lm/ngram\_language\_model.ipynb 참고

# STEP2: BOS, EOS 추가하기

>>> BOS = '<s>'

>>> EOS = '</s>'

>>> n = 2

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

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

>>> sentences

['<s> she sells sea-shells by the sea-shore. </s>', "<s> the shells she sells are sea-shells, i'm sure. </s>", "<s> for if she sells sea-shells by the sea-shore then i'm sure she sells sea-shore shells. </s>"]

<코드 끝>

이제 각 문장을 토큰화해서 하나의 큰 리스트로 만들자. 토큰화는 띄어쓰기(' ')를 기준으로 토큰화했다. 토큰화를 거치면 하나의 문장은 여러 개의 토큰으로 구성된 리스트가 된다. 토큰으로 이루어진 여러 개의 리스트를 하나의 큰 리스트로 합치는 과정은 [코드3]과 같이 reduce를 이용해 구현했다.

<코드 시작>

코드3: 토큰화 하기

# chapter1/ngram\_lm/ngram\_language\_model.ipynb 참고

# STEP3: 토큰화하기

>>> from functools import reduce

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

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

>>> sentences

[['<s>', 'she', 'sells', 'sea-shells', 'by', 'the', 'sea-shore.', '</s>'], ['<s>', 'the', 'shells', 'she', 'sells', 'are', 'sea-shells,', "i'm", 'sure.', '</s>'], ['<s>', 'for', 'if', 'she', 'sells', 'sea-shells', 'by', 'the', 'sea-shore', 'then', "i'm", 'sure', 'she', 'sells', 'sea-shore', 'shells.', '</s>']]

>>> tokens

['<s>', 'she', 'sells', 'sea-shells', 'by', 'the', 'sea-shore.', '</s>', '<s>', 'the', 'shells', 'she', 'sells', 'are', 'sea-shells,', "i'm", 'sure.', '</s>', '<s>', 'for', 'if', 'she', 'sells', 'sea-shells', 'by', 'the', 'sea-shore', 'then', "i'm", 'sure', 'she', 'sells', 'sea-shore', 'shells.', '</s>']

<코드 끝>

마지막으로 [코드4]에서 단어가 출현한 빈도가 한 번 이하인 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: 한번 출현한 단어 UNK으로 치환하기

# chapter1/ngram\_lm/ngram\_language\_model.ipynb 참고

# 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]

>>> tokens

['<s>', 'she', 'sells', 'sea-shells', 'by', 'the', '<unk>', '</s>', '<s>', 'the', '<unk>', 'she', 'sells', '<unk>', '<unk>', "i'm", '<unk>', '</s>', '<s>', '<unk>', '<unk>', 'she', 'sells', 'sea-shells', 'by', 'the', 'sea-shore', '<unk>', "i'm", '<unk>', 'she', 'sells', 'sea-shore', '<unk>', '</s>']

<코드 끝>

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

<코드 시작>

코드5: 문장 전처리 함수

# chapter1/ngram\_lm/ngram\_language\_model.ipynb 참고

import nltk

from functools import reduce

def preprocess(sentences, n):

'''문장으로 구성된 리스트를 쪼개서 토큰 리스트로 만듬

Args:

sentences (list of str): 여러 개의 문장으로 구성된 리스트

n (int): N-gram 모델의 N 계수

Returns:

토큰 리스트

'''

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>

1.2.2. 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: laplace 수식

<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. [코드4]에서 구한 tokens를 활용해서 [코드6]을 작성해보자.

<코드 시작>

코드6: N-gram 개수 세기

# chapter1/ngram\_lm/ngram\_language\_model.ipynb 참고

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

>>> vocab = nltk.FreqDist(bigram)

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

>>> a, b = k

>>> print(f'{a},{b}: {v}')

<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`의 bigram vocab을 만들고 그것을 표로 나타내면 아래와 같다.

<표 시작>

표1

표2

표3

chapter1.xlsx 참고

<표 끝>

[표1]에서 the는 총 3번 나왔다. the에 대한 bigram 카운트를 [표2]에서 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 구할 수 있다. 가령 P(she,sells|sells)의 확률은 0.3571이다. 이는 (4+1)/(4+10)이며 분자의 4는 she,sells의 빈도이고 분모의 4는 sells의 빈도이다. 그리고 분모에 더해주는 10은 총 고유 단어의 수이다.

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>

Add-k smoothing

수식5: Add-k smoothing

<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 [블록4] 같이 3-gram 언어 모델을 이용해서 문장의 확률을 구했다고 가정해보자.

<블록 시작>

블록4: 3-gram을 이용한 확률 계산시 Backoff 이용하기

아래의 확률은 실제 계산된 확률이 아닌 예시이다.

<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>|me) = 0.06

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

= P(love|<s>,I) P(you|I,love) \* P(because|love,you) \* P(you|you,because) \* P(love|because,you) \* P(me|you,love) \* P(</s>|love,me)

≒ P(love|<s>,I) P(you|I,love) \* P(because|love,you) \* P(you|you,because) \* P(love|because,you) \* P(me|you,love) \* P(</s>|me)

= 0.1 x 0.01 x 0.02 x 0.05 x 0.03 x 0.09 x 0.06

<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. [수식6]은 Backoff의 수식을 나타내고 있다.

<수식 시작>

수식6: Backoff 수식

<수식 끝>

Interpolation은 N-gram, (N-1)-gram, (N-2)-gram, ...을 일정 비율로 곱한 것이다. Interpolation의 수식은 [수식7]을 참고하면 된다.

<수식 시작>

수식7: 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.

1.2.3. N-gram 모델 학습하기

전처리된 토큰들을 이용해서 N-gram 모델을 만드는 SimpleNgramLanguageModel 클래스를 구현해보자. 이 클래스는 Unigram일 때와 그렇지 않을 때로 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: bigram vocabulary 만들기

>>> 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>

SimpleNgramLanguageModel 클래스의 build\_model 함수를 통해서 각 토큰에 대한 확률을 구할 수 있다. build\_model 함수는 [코드8]을 참고하라.

<코드 시작>

코드8: N-gram 모델 만들기

# chapter1/ngram\_lm/language\_model.ipynb 참고

def build\_model(self, tokens, n):

ngrams = nltk.ngrams(tokens, n)

nvocab = nltk.FreqDist(ngrams)

if n == 1:

vocab = nltk.FreqDist(tokens)

vocab\_size = len(nvocab)

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

else:

mgrams = nltk.ngrams(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.

1.2.4. 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. 1.2.2절에서 설명한 제로 카운트 해결하는 방법인 Laplace smoothing, Backoff, Interpolation 등의 기법이 나온 이유 역시 이러한 문맥의 부재를 조금이나마 덜기 위해서다. Backoff를 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. 가장 초창기 딥러닝 기반의 언어 모델 중 하나로 퍼셉트론(Perceptron)을 통해 학습한 Word2Vec을 들 수 있다.

1.3. Word2Vec 기반의 언어 모델

Word2Vec은 퍼셈트론을 이용해서 학습된 딥러닝 기반의 언어 모델이다. 퍼셉트론은 인공신경망 알고리즘 중 하나로 1957년에 코넬 대학교에서 프랭크 로젠블라트가 제안한 알고리즘이다. 퍼셉트론은 뇌를 구성하는 신경세포인 뉴런의 행동을 표현한 알고리즘이다. 뉴런은 서로 서로 연결돼 있다. 신호를 입력 값으로 받아들여서 역치 이상의 신호들을 다음 뉴런으로 전달하게 한다. 이 과정을 행렬 곱과 활성화 함수를 통해서 표현한 것이 퍼셉트론이다.

<그림 시작>

그림1: 행렬 연산으로 표현한 퍼셉트론

<그림 끝>

[그림1]을 보면 퍼셉트론 연산은 행렬 곱 연산으로 표현되며 행렬 곱 연산의 결과가 활성화 함수를 통해서 0과 1 사이의 값으로 출력이 결정된다는 것을 알 수 있다. 그리고 그 출력 값은 또 다른 퍼셉트론의 입력으로 사용된다.

Word2Vec은 퍼셉트론을 이용해서 학습한 언어 모델인데, 이 때 사용된 기법이 CBOW 또는 Skipgram이 있다.

<그림 시작>

그림2: CBOW, Skip-gram 학습 구조

<그림 끝>

[그림2]에서 CBOW는 Continuous Bag of Words의 약자이며 주변 단어를 이용해서 하나의 단어를 예측하는 방식이다. Skipgram은 CBOW와 반대로 하나의 단어를 이용해서 주변 단어를 예측하는 방식이다.

이렇게 학습한 Word2Vec 모델은 단어간의 관계를 벡터로 표현할 수 있다는 점에서 획기적인 성과를 보였다.

<그림 시작>

그림3: Word2Vec을 이용한 단어의 관계

남자:여자 = 왕:여왕

<그림 끝>

Word2Vec을 통해서 남자와 여자의 관계가 왕과 여왕의 관계와 유사하다는 것을 표현할 수 있다. 이전의 확률 기반의 언어 모델에서는 문맥을 표현하는 것이 불가능했다. 단순히 앞/뒤 단어의 빈도를 이용해서 확률 값을 표현했을 뿐이었다. 하지만 딥러닝 기반의 Word2Vec부터는 언어 모델이 단어를 이해할 수 있기 시작했다.

Word2Vec 모델의 한계는 단어와 단어와의 관계만 표현한다는 것이다. Word2Vec으로는 문맥을 표현할 수 없다. Word2Vec에서는 한 문장이 연속적인 단어로 이루어져 있을때 앞 단어로 무엇이 나왔는지에 따라 뒷 단어의 확률이 변하지 않는다. 한 단어에 대한 벡터 값이 고정돼 있기 때문이다. 이러한 문제점을 해결하기 위해 자연어 처리에서는 RNN 구조를 많이 사용해왔다.

1.4. RNN 기반의 언어 모델

1.4.1. RNN의 구조

RNN(Recurrent Neural Network)은 딥러닝을 이용해서 자연어 처리를 할 때 전통적으로 가장 오랫동안 사용됐었던 구조이다. RNN은 이전 단계의 출력을 되사용하는 순환적인 구조를 가지고 있다. RNN의 구조를 살펴보자.

<그림 시작>

그림4: RNN의 순환적인 구조

<그림 끝>

RNN의 입출력을 정의해보자. RNN은 입력 값(x)와 히든 스테이트(h)를 입력으로 사용해서 출력(y)을 만들어낸다. [그림4]에서 보면 RNN이 x0과 h0을 RNN의 입력으로 넣어서 y0을 출력했고, x1과 h1을 사용해서 y1을 출력했다. 이와 같이 이전 단계의 히든 스테이트 벡터를 다음 단계의 입력과 같이 사용해서 출력을 만들어내는 구조가 RNN이다. RNN의 내부 구조는 선형 연산과 활성화 함수 연산으로 이루어져 있다.

<그림 시작>

그림5: RNN 구조

<그림 끝>

[그림5]에서 RNN의 구조를 설명하고 있다. [그림5]에서 히든 벡터 h(t)와 y(t)를 만드는 공식을 수식으로 표현하면 [수식7]과 같다.

<수식 시작>

수식7: h(t)와 y(t)를 계산하기 위한 RNN 내부 수식

a(t) = tanh(Whh·h(t-1) + Whx·x(t-1) + bh)

y(t) = tanh(Wyh·a(t) + by)

<수식 끝>

[수식7]을 보면 입력 벡터와 히든 스테이트 벡터를 행렬 곱 연산 후 더해서 활성화 함수 tanh의 입력으로 넣는다. 이렇게 새로운 히든 스테이트 벡터 a(t)를 만들어 낸다. 그리고 a(t)를 다시 행렬곱 후 tanh로 활성화시켜서 y(t)를 구한다. 이 과정을 그림으로 표현하면 [그림5]와 같다.

RNN의 핵심은 이전 단어의 정보를 히든 스테이트로 표현해서 그 히든 스테이트를 다음 단어에 대한 히든 스테이트를 구할 때 재사용되는 것에 있다. 즉 x(t)를 처리할 때 x(t-1)에서 만들어진 h(t-1)이 영향을 주기 때문에 연속적인 데이터를 처리하는데 적합하다는 것이다. 하지만 RNN은 긴 시퀀스에 대한 정보를 저장하는 것에 대한 구조적인 문제가 있다.

<그림 시작>

그림6: RNN의 장기의존성 문제

<그림 끝>

[그림6]을 보면 x(0)을 표현하는데 h(0)이 사용됐다. 히든 스테이트 h(0)은 고정된 사이즈의 벡터일 것이다. h(1)은 x(0)과 x(1)을 표현하는데 사용되고 있다. h(1)역시 h(0)과 같은 크기인 고정된 사이즈의 벡터이다. 같은 원리로 x(0)부터 x(t)까지를 표현하기 위해서 사용되는 벡터는 h(t)이고 이 벡터의 사이즈는 h(0)의 사이즈와 같다. 더 많은 단어를 표현해야하는데 사용하는 벡터의 사이즈는 고정돼 있다는 것이 문제다. 결국 길이기 길어지면 길어질수록 이전 단어에 대한 표현은 점점 줄어들게 된다. 이것이 RNN이 가지고 있는 구조적인 문제점이며 장기의존성(Long-term dependancy) 문제라고 한다.

이 문제를 해결하기 위해서 LSTM이나 GRU가 등장했다. LSTM이나 GRU 모두 RNN을 기본으로 한 구조이기 때문에 이전의 히든 스테이트 값을 재사용하는 구조를 가지고 있다. 하지만 이전의 히든 스테이트 값을 선택적으로 사용할 수 있도록하는 구조가 추가돼 있다. LSTM의 경우 Forget Gate라는 구조를 추가하는데, Forget Gate를 통해서 이전의 정보를 잊을지 말지를 활성화 함수를 통해서 결정한다.

<수식 시작>

수식8: LSTM의 Forget Gate 내부 수식

f(t) = sigmoid(Wxf·x(t) + Whf·h(t-1) + bf)

<수식 끝>

[수식8]은 Forget Gate에 대한 수식이다. sigmoid함수는 0과 1사이의 값을 리턴하기 때문에 [수식8]의 연산을 통해서 이전 단계의 정보를 잊을지 말지 결정할 수 있다. GRU에서는 이러한 역할을 Update Gate가 담당한다. GRU의 Update Gate는 이전과 현재의 히든 스테이트 값을 어느 정도의 비율로 반영할지를 결정한다. 다음 절에서는 GRU를 이용해서 간단하게 언어 모델을 구현해보고자 한다.

1.4.2. GRU 언어 모델 구현하기

이번 절에서는 GRU를 이용해서 간단한 언어 모델을 구현해보려고 한다. 잘 동작하는 언어 모델을 만드는 것은 매우 어려운 일이다. 따라서 이번 절에서의 예시는 GRU가 입력 데이터를 얼마나 잘 생성하는지를 보여주는 정도로 이해하면 된다.

<그림 시작>

그림7: 시퀀스에 대한 GRU 입출력

<그림 끝>

GRU 언어 모델에 대한 입력과 출력을 정의해보자. [그림7]을 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 [블록5]와 같이 정리할 수 있다.

<블록 시작>

블록5: 토큰 시퀀스에 대한 GRU 입출력

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. 레포지토리에서 chater1/gru\_lm/gru\_language\_model.ipynb을 참고하라.

<코드 시작>

코드9: GRU 언어 모델을 위한 데이터셋 클래스 정의하기

# chapter1/gru\_lm/dataset.py 참고

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>

[코드9]의 GRULanguageModelDataset은 아래와 같이 실행할 수 있다.

<코드 시작>

코드10: GRU 언어 모델을 위한 데이터셋 객체 만들기

# chapter1/gru\_lm/gru\_language\_model.ipynb 참고

>>> 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>

[코드10]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.

<팁 시작>

torch를 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.

<팁 끝>

그 다음에는 DataLoader를 만들 차례이다. dataset을 이용해서 [코드11]와 같이 만들면 된다.

<코드 시작>

코드11: GRU 언어 모델을 위한 데이터로더 만들기

# chapter1/gru\_lm/gru\_language\_model.ipynb 참고

>>> 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>

torch를 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 [코드12]과 같이 (batch\_size, max\_length) shape의 텐서를 지속적으로 Feeding한다.

<코드 시작>

코드12: GRU 언어 모델 학습하기 전 데이터 로더 테스트하기

# chapter1/gru\_lm/gru\_language\_model.ipynb 참고

>>> 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. [코드13]와 같이 간단하게 만들 수 있다.

<코드 시작>

코드13: GRU 언어 모델 정의하기

# chapter1/gru\_lm/model.py 참고

from torch import nn

class GRULanguageModel(nn.Module):

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

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

self.hidden\_size = hidden\_size

self.output\_size = output\_size

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\_size에 따른 모델 학습 결과의 차이는 1.4.3절에서 비교할 예정이다. 데이터 전처리와 Feeding 그리고 모델링까지 살펴봤다. 이제 모델을 학습하는 부분을 [코드14]를 통해 살펴보자.

<코드 시작>

코드14: GRU 언어 모델 학습을 위한 함수 정의하기

# chapter1/gru\_lm/run\_gru.py 참고

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>)

'''

hidden\_size = model.hidden\_size

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 때 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>

그림8: Teacher-forcing 개념

<그림 끝>

[그림8]를 보면 teacher forcing 기법을 사용할 경우와 그렇지 않은 경우를 잘 표현하고 있다. teacher forcing의 경우 GRU에 들어가는 입력이 label에서 온다. 처음 시작인 I만 Input에서 오고 그 다음 love, you는 label에서 입력된다. 출력을 보면 label과 전혀 다른 love, him, but 등을 출력하고 있지만 그것들을 학습에 사용하지 않고 학습에는 제대로된 정답인 love, you, </s>를 사용하는 것이 teacher forcing이다. 반면에 love, him 등을 다음 GRU의 입력으로 사용하는 경우가 teacher forcing 기법을 사용하지 않는 경우이다.

그 다음으로 train 함수에서 설명할 부분은 max\_grad\_norm 부분이다. 이 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 gradient의 l2norm 값으로 나눠고 threshold 값을 곱해주는 것이다. [수식5]를 참고하자.

<수식 시작>

if |g| > threshold

gradient = (threshold/l2norm) \* gradient

else

gradient = gradient

수식5: gradient clipping 적용하기

<수식 끝>

train 함수 사용 예시는 [코드14]에 주석으로 설명돼 있는 Example을 참고하면 된다. run\_gru.py를 실행하면 전체 학습 과정을 실행할 수 있다. run\_gru.py를 실행하면 [코드15]과 같이 출력을 한다.

<코드 시작>

코드15: GRU 언어 모델 학습 스크립트 실행하기

$ python run\_gru.py --hidden\_size=30

0th iteration -> loss=58.4429

0th iteration -> loss=60.8093

0th iteration -> loss=59.7647

1th iteration -> loss=57.4985

1th iteration -> loss=59.2246

1th iteration -> loss=56.9591

2th iteration -> loss=56.2083

...

998th iteration -> loss=5.0649

999th iteration -> loss=19.3040

999th iteration -> loss=4.0943

999th iteration -> loss=17.3739

<코드 끝>

run\_gru.py에서는 chapter1/gru\_lm/input\_data.txt를 학습한다. 이 데이터는 "she sells sea shells by the sea shore. the shells she ~"로 반복되는 텍스트 데이터이다. 이 데이터를 학습 데이터로 사용한 이유는 이 책을 학습할 때 수월하게 하기 위해서이다. 보통 언어 모델을 학습하려면 고성능의 컴퓨팅 리소스를 필요로 한다. 하지만 공부를 목적으로 고성능의 컴퓨팅 리소스를 반드시 사용해야 한다면 그것은 효율적이지 않다. 따라서 일반적인 노트북 사양에서도 동작하는 언어 모델을 만들어 볼 수 있도록 데이터의 양과 복잡성을 크게 제한했다.

1.4.3. GRU 언어 모델로 문장 생성하기

마지막으로 <s>부터 문장을 생성해보자. 문장을 생성하는 generate\_sentence\_from\_bos 함수는 [코드16]과 같이 구현할 수 있다.

<코드 시작>

코드16: 학습된 GRU 언어 모델을 이용해 문장 생성하기

# chapter1/gru\_lm/generate.py 참고

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 [블록6] 같은 문장을 생성하게 된다.

<블록 시작>

블록6: GRU 언어 모델을 이용해서 생성한 문장

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

<블록 끝>

마지막으로 [블록7]에서는 hidden\_size 변화에 따른 모델의 문장 생성 능력을 비교하고 있다. hidden\_size=1과 hidden\_size=30으로 각각 모델링을 한 후 학습된 모델로 문장을 생성하면 [블록7]과 같이 큰 차이를 나타낸다. hidden\_size 외의 다른 조건은 동일하게 학습했다.

<블록 시작>

블록7: hidden\_size에 따른 GRU 언어 모델의 성능 차이 비교

# hidden\_size = 1로 학습한 경우,

# $ python run\_gru.py --hidden\_size=1

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,

# $ python run\_gru.py --hidden\_size=30

generated\_sentence: <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.