2. 집중해보자! 어텐션

<블록 시작>

윤우아빠는 다음 달에 미국으로 가족 여행을 떠나기로 했다.

윤우는 신나게 들떠 있는데 영어를 할 줄 모르는 윤우아빠는 심각하기만 하다.

그날 저녁부터 여행을 가기 위한 영어 공부를 시작하려는데

우리 말이랑 단어의 순서도 다르고 한번도 본적도 없는 알파벳들이 혼란스럽기만 하다.

<블록 끝>

한번도 배운 적 없는 언어를 배우는 것은 어렵다. 사람이 새로운 언어를 배우려면 단어도 외워야 하고, 문법도 공부해야 하고, 책도 많이 읽어야 한다. 언어 능력은 하루 아침에 습득할 수 없다. 시간이 오래 걸린다. 하지만 한번 배워두면 머리 속에 오래 남고 그 언어로 된 만화책을 읽든 소설을 읽든 뉴스를 보든 상관없이 활용할 수 있다. 즉 우리는 어렸을 때부터 많은 단어를 듣고 쓰고 읽고 말하면서 언어 능력을 취득했고, 그 능력은 언어가 필요한 곳이라면 어떤 곳에도 응용해서 쓸 수 있다. 이렇게 필요한 곳이라면 어떤 곳에도 응용을 할 수 있는 이유는 사람은 언어의 문맥을 이해하고 있기 때문이다.

자연어 처리도 컴퓨터로 하여금 언어를 이해하게 하여 사람이 자연어를 처리하는 것과 같이 처리할 수 있도록 구현하는 것이다. 컴퓨터가 언어를 이해할 수 있게 만들기 위해 필요한 것이 언어 모델이고, 2장에서 N-Gram 언어 모델과 딥러닝 기반의 언어 모델 두 가지를 살펴봤다. N-Gram 언어 모델은 단순히 단어를 확률적으로 나타냈을 뿐이다. 문맥을 이해하고 있지 않는 언어 모델이다. 딥러닝의 RNN 구조의 언어 모델은 언어의 문맥을 이해할 수 있는 구조를 가지고 있다. RNN 구조의 언어 모델은 하나의 문장을 은닉 벡터로 표현할 수 있다.

2.1. 하나의 벡터로 모든 정보를 담는 RNN

RNN 구조의 언어 모델부터는 문맥을 이해할 수 있는 능력을 가진 언어 모델이다. 그런데 RNN의 특성 상 크게 두 가지 문제점이 있다. 하나는 장기 의존성(Long-Term Dependency) 문제이다. RNN은 기본적으로 이전의 출력을 그 다음의 입력으로 넣는 Auto-Regressive한 특성을 가지고 있기 때문에 그 과정이 반복되면 될수록 앞에서 입력됐던 단어에 대한 기억은 점점 희미해지게 된다. 또 다른 문제는 하나의 은닉 벡터가 모든 정보를 담는다는 것이다. RNN의 Auto-Regressive한 특성 때문에 은닉 벡터는 모든 단어를 하나의 은닉 벡터에 표현하려고 한다.

5평짜리 방에서 사는 것을 상상해보자. 혼자 산다고 가정하면 그래도 어느 정도 괜찮을 것이다. 그런데 친구 5명이서 그 방에서 산다면 어떨까? 잠을 잘 때도 서로 엉겨붙어서 자야 할 것이다. RNN은 입력 문장의 총 길이에 상관없이 그 입력을 고정된 길이의 벡터로 인코딩한다. 입력문장이 길어지면 길어질수록 많은 정보를 벡터에 넣어야 하기 때문에 인코딩이 힘들어질 수 밖에 없다. 실제로 [그림1]과 같이 문장의 길이가 길어질수록 LSTM의 성능이 떨어진다는 연구결과가 있다.

<그림 시작>

그림1

[그림1: 시퀀스 길이에 따른 BLEU 스코어 변화]

BLEU 스코어: 번역 시스템을 평가하는데 사용하는 평가 방법으로 높을수록 좋은 성능을 갖는다.

https://arxiv.org/pdf/1409.1259.pdf 참고

<그림 끝>

조금 더 구체적인 예를 들어보자. RNN으로 번역 시스템을(NMT) 만든다고 가정해보자. NMT는 기본적으로 입력 문장을(Sequence) 번역문(Sequence)으로 바꿔주는 Sequence-To-Sequence(Seq2Seq) 구조를 갖게 된다. Seq2Seq는 인코더와 디코더 구조로 이루어져 있다. 인코더는 입력 문장의 각 단어를 하나하나 RNN으로 인코딩한다. 디코더는 인코더의 결과를(encoded) 활용하여 번역문을 구성하는 단어를 하나 하나 만드는 것이다.

<그림 시작>

그림2

[그림2: Seq2Seq 구조]

<그림 끝>

Seq2Seq 구조의 모델이 훌륭한 번역기로 학습되려면 [그림2]의 encoded vector가 입력문장의 모든 단어를 표현하고 있어야 한다. 짧은 문장을 번역할 때는 번역할 문장의 정보를 encoded vector에 충분히 담을 수 있지만 긴 문장에 대한 번역을 진행할 때는 encoded vector의 크기가 고정돼 있기 때문에 짧은 문장보다 정보를 잘 표현할 수 없게 된다. 이러한 장기 의존성 문제를 해결하기 위해 고안된 방법이 어탠션 매커니즘이다.

2.2. 왜 어텐션(Attention)하지 않지?

앞 절에서 RNN은 입력 데이터를 하나의 벡터로 표현하는 구조를 가지고 있고, 이러한 특성 때문에 입력 데이터가 길어질 경우 입력을 하나의 벡터로 표현하기가 힘들어진다는 것을 설명했다. 그렇다면 어떻게 이 문제를 해결할 수 있을까? 필요한 부분만 집중해서(Attention) 보면 된다. 번역기에서 입력 문장을 번역문으로 바꿀 때 각 입력 단어를 해석할 때 집중해야 되는 부분이 정해져 있다. 아래의 빈 칸 채우기 문제를 봐보자.

ENG: My name is Dooli.

KOR: 내 이름은 둘리\_\_.

빈 칸에 들어갈 정답은 "입니다", "이다" 등이 적절하다. 저 빈 칸을 채우기 위해서 "My name" "Dooli" 등에 집중할 이유가 있을까? 물론 부분적으로 있지만, 가장 집중해야 되는 부분은 "is"이다. RNN에서 집중을 해야하는 특정 부분에만 집중하도록 weight을 주는 것이 바로 어텐션 메커니즘이다. [그림2]에 어텐션 네트워크를 추가하면 [그림3]과 같은 구조가 된다.

<그림 시작>

그림3

[그림3: Attention이 추가된 Seq2Seq 구조]

<그림 끝>

디코더에서 번역 단어를 하나 하나 출력할 때마다 어텐션 가중치를 추가해주는데, 이 어텐션 가중치는 인코더의 output 정보를 통해서 계산된다. [그림2]와 [그림3]을 비교해보자. [그림2]에서 디코더를 연산할 때 인코더의 은닉 스테이트 값만 들어간다. 하지만 [그림3]에서는 인코더의 값에서 중요한 부분에 가중치를 주는 부분이 추가됐다. 그 가중치를 통해서 컨텍스트 벡터를 만들었고 컨텍스트 벡터와 인코더의 아웃풋을 이용해서 어탠션이 적용된 백터(attention-applied vector)를 만들었다. 이 벡터를 디코더에서 이용하게 된다. 어텐션 네트워크에 대한 자세한 구조는 다음 절에서 자세하게 설명할 예정이다.

2.3. 어떻게 어텐션(Attention)하지?

이전의 절에서 어텐션을 왜 해야하는지에 대한 직관적인 이해를 해봤다. Seq2Seq 구조는 인코더와 디코더의 구조로 이루어져 있는데, 시퀀스의 길이가 길어질수록 인코더에서 시퀀스를 효과적으로 표현할 수 없기 때문에 무언가 플러스 알파가 필요한 것이다. 그것이 어텐션이며, 그 어텐션은 인코더의 값을 참고해서 가중치를 만든 후 디코더의 값을 만들어가는 것이다.

이번 절에서는 어텐션에 대한 구체적인 이야기를 해보려고 한다. 어텐션을 공부할 때 이해해야 하는 쿼리, 키, 값(Query, Key, Value)가 무엇인지 그리고 그것을 이용해서 어떻게 어탠션을 구현할 수 있는지 알아보자.

2.3.1. 묻고 참고하고 답하기

<박스 시작>

영어 울렁증이 심한 윤우아빠가 길을 걸어가고 있는데, 갑자기 파란 눈의 외국인이 말을 걸었다.

외국인: I want to go to the N-Tower. Where can I take a bus?

윤우아빠: 음.... 엔타워 버스 스테이션... 인프론트오브 맬도날드... 쓰리투제로 버스...

<박스 끝>

영어 울렁증이 심한 윤우아빠에게 언제라도 있을 수 있는 일이다. 외국인이 유창한 영어로 물어봐도 윤우아빠 귀에는 몇 단어 들리지 않았을 것이다.

<그림 시작>

그림4

[그림4: 윤우아빠 머리속 단어들 - 쿼리]

<그림 끝>

윤우아빠는 이 몇 단어를 조합해서 머리 속에 있는 몇 개 없는 영어 단어와 매칭해서 외국인이 뭘 원하는지 이해하려고 쩔쩔매고 있었을 것이다.

<그림 시작>

그림5

[그림5: 단어 키-값]

<그림 끝>

몇 개 알아들은 단어들을 조합해서 윤우아빠는 드디어 외국인이 원하는 그림을 머리속으로 그려낸다.

<그림 시작>

그림6

[그림6: 컨택스트 그림]

<그림 끝>

이것이 쿼리, 키, 값의 기본적인 개념이다. 쿼리는 외국인이 말한 문장에서 들은 몇 개의 단어들(N-Tower, go, bus, where)이다. 그리고 윤우아빠가 외우고 있는 영어-한국어 단어가 각각 키, 값류이다. 문법을 모르는 윤우아빠는 자기가 들은 몇 개의 단어만을 조합해서 상황을 판단해야 하는데, 최대한 자기가 이미 알고 있는 것과 유사한 조합으로 상황을 이해하려고 노력하게 된다.

<그림 시작>

그림7

[그림7: 확률그림]

<그림 끝>

[그림7]을 보면 리스닝한 영어 단어마다 연상되는 영어 단어를 확률적으로 매칭했고, 윤우아빠는 그 모든 결과를 조합해서 드디어 그 상황(컨택스트)를 이해했다. 다음 절에서는 어탠션 매커니즘을 조금 더 수학적으로 접근해서 실제로 어탠션을 계산하는 방법을 알아보자.

2.3.2. 어텐션 계산해보기

앞 절에서 예시를 통해 어텐션을 이해해봤다. 그런데 이 책에서는 AI를 기숙적인 관점에서 공부하고 있기 때문에 조금 더 구체적인 이해를 해야한다. 어텐션을 한마디로 표현하면 내가 찾고자 하는 것을(쿼리) 내가 아는 지식(키,값)을 이용해서 찾아내는 것이다. 쿼리, 키, 값이 어텐션에서 어떻게 계산되는지 알아보자.

어텐션을 구하는 과정은 쿼리와 키의 유사도를 구하는 과정이 핵심이다. 쿼리와 키의 유사도를 구한 후 그 유사도에 Softmax 등을 취해서 합이 1인 비율로 바꿔주는 것이다. [그림7]에서 예시로 사용된 확률 값이 사실은 Softmax를 이용해서 만들어진 값이라고 이해하면 된다.

<수식 시작>

수식1

유사도(쿼리, 키) = qk

weight = softmax(qk)

context = weight \* 값

<수식 끝>

[수식1]에서 쿼리는 디코더의 입력이다. 키는 인코더 RNN의 아웃풋을 사용하고, 값의 경우는 키와 같은 값을 사용한다. 유사도를 이용해서 qk를 만들고 softmax 함수를 이용해서 weight을 만든다. 여기까지가 어텐션을 만드는 과정이다. 이렇게 만든 weight을 값에 곱해주는 과정이 어텐션을 적용하는 과정이 되고 그 결과를 context라고 한다. 유사도를 구하는데 사용하는 함수는 [표1]을 참고하라.

<표 시작>

표1

chapter2/table1

어탠션 alignment 함수 종류

<표 끝>

이 책에서는 [표1]에서 행렬을 곱하는 방법을 유사도 측정 함수로 정하고 이야기해보자. 왜 유사도를 측정하기 위해서 행렬의 곱셈을 이용하는 것일까? 행렬 곱셈을 하는 방법이 곧 행렬의 내적을 구하는 방법이고 행렬의 내적은 cosine을 이용해서 표현할 수 있기 때문이다. [수식2]을 보면 두 개의 벡터 a와 b가 있을 때 그 벡터의 사잇각을 구하는 방법이 결국 행렬의 곱으로 이루어진다는 것을 알 수 있다. 쿼리와 키도 결국에는 벡터이다. 쿼리와 키의 유사도를 구한다는 것은 결국 두 벡터의 유사도를 구하는 것과 같고, 유사도를 구하는 방법은 [표1]과 같이 여러가지 방법이 있는데 이 책에서는 행렬의 곱셈을 이용한 방법으로 유사도를 구하려고 한다.

<수식 시작>

수식2

a\_vec \* b\_vec = |a\_vec| |b\_vec| cosine

a\_vec \* b\_vec = 원소끼리 곱하기 → i.e. 행렬곱

위 두 식을 서로 같게 두고 전개해서

cosine = 행렬곱 / (|a\_vec| |b\_vec|)

<수식 끝>

행렬의 곱셈을 이용해서 유사도를 구한 후 그 유사도를 값에 곱해주면 된다.

## 쿼리,키,값을 형광팬 하이라이트했으면 합니다.

결국 어텐션을 구하는 과정은 쿼리와 키간의 유사도를 구하는 과정과 그 유사도를 값에 곱해주는 과정으로 이루어지는 셈이다.

어탠션을 구하는 공식을 조금 이론적으로 접근해서 알아봤다. 이제는 실제 벡터 연산을 통해서 어탠션 메커니즘을 계산해보자.

<그림 시작>

그림8

<그림 끝>

문장 "I love you so much"과 문장 "나는 너를 너무 사랑해"의 어탠션을 계산해보려고 한다. 각 문장은 다섯 개의 토큰으로 쪼개져있고 각 토큰의 벡터는 [그림8]와 같다.

<그림 시작>

그림9

<그림 끝>

[그림9]는 [그림8]의 벡터 값을 이용해서 어탠션 가중치를 구하고 있다. 어탠션 가중치는 행렬 곱셈과 소프트맥스 함수를 통해서 구할 수 있다. [그림9]에서의 연산 결과를 보면 문장 "I love you so much"와 문장 "나는 너를 너무 사랑해"의 어탠션 가중치를 토큰 별로 알 수 있다. 예를 들어서 토큰 "love"에 대해서 토큰 "사랑"에 대한 가중치는 0.222이다.

## 각주1 [그림8]과 [그림9]에서의 토큰 벡터는 설명을 위해 편이상 임의로 정한 벡터이기 때문에 어탠션 가중치가 문맥에 맞게 나오지 않았다. 실제 학습이 제대로 진행된 경우에는 어탠션 가중치가 문맥에 맞게 중요한 부분과 그렇지 않은 부분을 잘 구분한다.

<그림 시작>

그림10

<그림 끝>

[그림9]에서 계산한 가중치 값을 값 벡터에 적용하면 [그림10]과 같은 결과를 얻을 수 있다.

이 절에서 어탠션을 계산하는 방법에 대해서 자세하게 다뤘다. 어텐션은 쿼리와 키의 유사도를 구한 후 그 유사도를 Softmax해서 합이 1인 형태의 가중치로 변형해서 만든다. 유사도를 구하는 다양한 방법은 [표1]에 소개돼 있다. 이렇게 만든 가중치를 값에 곱하면 컨택스트 벡터를 만들 수 있다. 즉 컨텍스트 벡터는 쿼리와 키의 유사도 가중치를 값 벡터에 적용한 벡터인 셈이다.

2.3.3. 어탠션 구현하기

이번 절에서는 어탠션을 구현해보자. 실제 번역기 등을 만들어볼 수도 있지만, 이번 절에서는 어탠션을 잘 이해할 수 있기 위해 간단하게 임의의 알파벳 시퀀스를 거꾸로 뒤집는 Seq2Seq 모델을 만들어보려고 한다. 문자열을 뒤집는 모델을 예시로 선택한 이유는 어텐션을 설명하기에 조금 쉽다고 생각했기 때문이다. 단순히 문자열의 순서를 뒤집는 모델이기 때문에 데이터에 대한 사전 지식이 필요하지 않고, 학습하는데도 그렇게 큰 컴퓨팅 자원이 필요하지 않기 때문이다. 구현하고자 하는 모델의 입력과 출력을 [블록1]을 통해서 확인해보자.

<블록 시작>

블록1

ex-1) thb → bht

ex-2) pqovqh → hqvoqp

ex-3) vnwqjkm → mkjqwnv

<블록 끝>

t를 입력으로 받았을 때 아웃풋으로 가장 집중해야 되는 부분이 b이다. h를 입력으로 받았을 때는 h이고 b를 입력으로 받았을 때는 t에 가장 집중해야 한다. 이제 구현을해보자. 이 책의 레포지토리 chapter2/attention-nmt.ipynb를 참고하면 된다.

우선 학습할 데이터셋을 확보하기 위해서 랜덤 알파벳 스트링을 생성하는 함수를 [코드1]과 같이 만들어보자.

<코드 시작>

코드1

def generate\_random\_alphabet\_index():

random\_length = np.random.randint(10, MAX\_LENGTH-2) # -2 because of <s> and </s>

random\_alphabet\_index = np.random.randint(0, 26, random\_length) + 3

return random\_alphabet\_index.tolist(), random\_length

<코드 끝>

위의 함수를 실행하면 아래와 같이 랜덤한 길이의 알파벳 인덱스 리스트와 그 길이를 얻을 수 있다.

<코드 시작>

>>> from dataset import generate\_random\_alphabet\_index

>>> generate\_random\_alphabet\_index()

([21, 6, 26, 23, 28, 15, 18, 4, 21, 15, 19, 19], 12)

>>> generate\_random\_alphabet\_index()

([13, 22, 19, 4, 10, 19, 25, 27, 6, 23, 25, 14, 18], 13)

<코드 끝>

보통 AI 모델에 입력을 넣을 때는 데이터의 길이가 고정이어야 한다. 여기에서는 최대 길이를 15로 정의하자. 그러면 최대 길이 15 이내에 대해서 </s>와 <pad> 등을 추가해줘야 한다. 이러한 작업을 AlphabetToyDataset 클래스를 통해서 해보자. [코드2]는 AlphabetToyDataset을 구현한 것이다.

<코드 시작>

코드2

from torch.utils.data import Dataset

# 데이터셋 만들기

class AlphabetToyDataset(Dataset):

def \_\_init\_\_(self, n\_dataset=1000):

bos = 0

eos = 1

pad = 2

self.inputs = []

self.labels = []

self.length = []

for \_ in range(n\_dataset):

# make input example

aindex, alen = generate\_random\_alphabet\_index()

# index to alphabet

#alphabet = list(map(lambda a: i2a[a], aindex))

# inversing

#inversed\_alphabet = list(map(lambda a: inverse\_map[a], alphabet))

# alphabet to index

#iindex = list(map(lambda ia: a2i[ia], inversed\_alphabet))

iindex = aindex[::-1]

# add bos, eos and pad

n\_pad = MAX\_LENGTH - len(aindex) - 1

aindex = aindex + [eos] + [pad]\*n\_pad

iindex = iindex + [eos] + [pad]\*n\_pad

# add to examples

self.inputs.append(aindex)

self.labels.append(iindex)

self.length.append(alen)

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

return len(self.inputs)

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

return [

torch.tensor(self.inputs[index], dtype=torch.long),

torch.tensor(self.labels[index], dtype=torch.long),

torch.tensor(self.length[index], dtype=torch.long)

]

<코드 끝>

위의 AlphabetToyDataset을 클래스는 generate\_random\_alphabet\_index 함수를 통해서 생성된 랜덤 알파벳 a부터 z까지를 각각 3부터 28까지 치환하여 inputs을 만들고, 그것을 역순으로 배열하여 labels를 만든 것이다. 즉, 랜덤하게 생성된 알파벳을 역순으로 배열하는 간단한 AI 모델을 만들기 위한 데이터셋이다.

AlphabetToyDataset을 [코드3]과 같이 사용해서 데이터셋을 만들수 있다.

<코드 시작>

코드3

train\_dataset = AlphabetToyDataset(n\_dataset=3000)

valid\_dataset = AlphabetToyDataset(n\_dataset=300)

<코드 끝>

AlphabetToyDataset으로 train\_dataset과 valid\_dataset을 만들고, 그것을 DataLoader를 이용해서 AI 모델에 입력할 수 있도록 배치 사이즈도 설정해줄 수 있다. [코드4]를 보자.

<코드 시작>

코드4

import torch

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

from torch.utils.data import DataLoader

def collate\_fn(batch):

inputs = pad\_sequence([b[0] for b in batch], batch\_first=True)

targets = pad\_sequence([b[1] for b in batch], batch\_first=True)

lengths = torch.stack([b[2] for b in batch])

lengths, indice = torch.sort(lengths, descending=True)

inputs = inputs[indice]

targets = targets[indice]

return inputs, targets, lengths

train\_dataloader = DataLoader(train\_dataset, collate\_fn=collate\_fn, batch\_size=16)

valid\_dataloader = DataLoader(valid\_dataset, collate\_fn=collate\_fn, batch\_size=1)

<코드 끝>

[코드5]에서 for문을 통해서 DataLoader가 내뱉어내는 데이터를 검토해보자.

<코드 시작>

코드5

for d in train\_dataloader:

inputs, targets, lengths = d

print(inputs)

print(targets)

print(lengths)

print(inputs.shape, targets.shape, lengths.shape)

break

tensor([[ 6, 24, 18, 24, 6, 16, 20, 27, 20, 18, 24, 19, 1, 2, 2],

[25, 14, 18, 19, 23, 7, 13, 28, 3, 24, 25, 21, 1, 2, 2],

[ 4, 7, 25, 18, 16, 17, 27, 27, 21, 10, 6, 19, 1, 2, 2],

[12, 9, 12, 16, 7, 20, 18, 19, 14, 18, 24, 1, 2, 2, 2],

[18, 16, 19, 14, 4, 18, 8, 4, 7, 10, 7, 1, 2, 2, 2],

[20, 7, 20, 20, 5, 7, 21, 20, 14, 10, 25, 1, 2, 2, 2],

[18, 25, 20, 7, 6, 24, 25, 22, 20, 14, 10, 1, 2, 2, 2],

[27, 3, 28, 28, 4, 18, 26, 28, 22, 21, 16, 1, 2, 2, 2],

[16, 25, 27, 15, 20, 25, 11, 10, 13, 14, 16, 1, 2, 2, 2],

[16, 14, 13, 3, 3, 20, 9, 14, 16, 3, 1, 2, 2, 2, 2],

[20, 26, 27, 23, 26, 15, 19, 24, 8, 23, 1, 2, 2, 2, 2],

[13, 27, 19, 21, 12, 11, 4, 11, 8, 21, 1, 2, 2, 2, 2],

[ 6, 28, 3, 27, 4, 14, 11, 23, 5, 22, 1, 2, 2, 2, 2],

[17, 16, 5, 8, 6, 12, 28, 17, 27, 14, 1, 2, 2, 2, 2],

[22, 26, 20, 19, 10, 8, 18, 17, 17, 10, 1, 2, 2, 2, 2],

[25, 20, 28, 13, 20, 4, 11, 19, 24, 5, 1, 2, 2, 2, 2]])

tensor([[19, 24, 18, 20, 27, 20, 16, 6, 24, 18, 24, 6, 1, 2, 2],

[21, 25, 24, 3, 28, 13, 7, 23, 19, 18, 14, 25, 1, 2, 2],

[19, 6, 10, 21, 27, 27, 17, 16, 18, 25, 7, 4, 1, 2, 2],

[24, 18, 14, 19, 18, 20, 7, 16, 12, 9, 12, 1, 2, 2, 2],

[ 7, 10, 7, 4, 8, 18, 4, 14, 19, 16, 18, 1, 2, 2, 2],

[25, 10, 14, 20, 21, 7, 5, 20, 20, 7, 20, 1, 2, 2, 2],

[10, 14, 20, 22, 25, 24, 6, 7, 20, 25, 18, 1, 2, 2, 2],

[16, 21, 22, 28, 26, 18, 4, 28, 28, 3, 27, 1, 2, 2, 2],

[16, 14, 13, 10, 11, 25, 20, 15, 27, 25, 16, 1, 2, 2, 2],

[ 3, 16, 14, 9, 20, 3, 3, 13, 14, 16, 1, 2, 2, 2, 2],

[23, 8, 24, 19, 15, 26, 23, 27, 26, 20, 1, 2, 2, 2, 2],

[21, 8, 11, 4, 11, 12, 21, 19, 27, 13, 1, 2, 2, 2, 2],

[22, 5, 23, 11, 14, 4, 27, 3, 28, 6, 1, 2, 2, 2, 2],

[14, 27, 17, 28, 12, 6, 8, 5, 16, 17, 1, 2, 2, 2, 2],

[10, 17, 17, 18, 8, 10, 19, 20, 26, 22, 1, 2, 2, 2, 2],

[ 5, 24, 19, 11, 4, 20, 13, 28, 20, 25, 1, 2, 2, 2, 2]])

tensor([12, 12, 12, 11, 11, 11, 11, 11, 11, 10, 10, 10, 10, 10, 10, 10])

torch.Size([16, 15]) torch.Size([16, 15]) torch.Size([16])

<코드 끝>

이 데이터를 이용해서 학습할 모델을 만들어보자. [코드6]을 참고하면 된다.

<코드 시작>

코드6

class EncoderRNN(nn.Module):

def \_\_init\_\_(self, input\_size, hidden\_size):

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

self.hidden\_size = hidden\_size

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

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

def forward(self, inputs, hidden):

'''

Input Parameters

- inputs: (B,M)

- hidden: (1,B,H)

Output returns

- output: (B,1,O)

- hidden: (B,1,H)

- attn\_weights: (B,1,M)

Logging outputs

\*\* output: torch.Size([16, 1])

\*\* hidden: torch.Size([16, 1])

'''

#print('\*\* inputs: {}'.format(inputs.shape))

#print('\*\* hidden: {}'.format(hidden.shape))

embedded = self.embedding(inputs) # (B,M,H)

#print('\*\* embedded: {}'.format(embedded.shape))

output, hidden = self.gru(embedded, hidden) # (B,M,H), (1,B,H)

return output, hidden

def initHidden(self, batch\_size):

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

# with attention

class AttnDecoderRNN(nn.Module):

def \_\_init\_\_(self, hidden\_size, output\_size, dropout\_p=0.1, max\_length=MAX\_LENGTH):

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

self.hidden\_size = hidden\_size

self.output\_size = output\_size

self.dropout\_p = dropout\_p

self.max\_length = max\_length

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

self.attn = nn.Linear(self.hidden\_size \* 2, self.max\_length)

self.attn\_combine = nn.Linear(self.hidden\_size \* 2, self.hidden\_size)

self.dropout = nn.Dropout(self.dropout\_p)

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

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

def forward(self, inputs, hidden, encoder\_outputs):

embedded = self.embedding(inputs) # (B,1,H)

embedded = self.dropout(embedded)

# query: embedded

# key: hidden

# value: encoder\_outputs

attn\_weights = F.softmax(

self.attn(

torch.cat((embedded, hidden.transpose(0, 1)), -1) # (B,1,2H)

), # (B,1,M)

dim=-1) # (B,1,M)

#print('\*\* attn\_weights: {}'.format(attn\_weights.shape))

#print('\*\* attn\_weights: {}'.format(attn\_weights))

#print('\*\* attn\_weights.sum(): {}'.format(attn\_weights.sum().shape))

#print('\*\* attn\_weights.sum(): {:.4f}'.format(attn\_weights.sum()))

attn\_applied = torch.bmm(attn\_weights, encoder\_outputs) # (B,1,H)

#print('\*\* attn\_applied: {}'.format(attn\_applied.shape))

#return

output = torch.cat((embedded, attn\_applied), -1) # (B,1,2H)

#print('\*\* output: {}'.format(output.shape))

output = self.attn\_combine(output) # (B,1,H)

#print('\*\* output: {}'.format(output.shape)) # (B,1,H)

output = F.relu(output)

#print('\*\* output: {}'.format(output.shape)) # (B,1,H)

#print('\*\* hidden: {}'.format(hidden.shape)) # (B,1,H)

output, hidden = self.gru(output, hidden) # (B,1,H) (1,B,H)

#print('\*\* gru-output: {}'.format(output.shape))

#print('\*\* gru-hidden: {}'.format(hidden.shape))

output = F.log\_softmax(self.out(output), dim=-1)

#print('\*\* final-output: {}'.format(output.shape))

#print('\*\* final-hidden: {}'.format(hidden.shape))

#print('\*\* final-attn\_weights: {}'.format(attn\_weights.shape))

return output, hidden, attn\_weights

def initHidden(self, batch\_size):

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

<코드 끝>

모델링 부분은 조금 더 자세하게 설명해보려고 한다. 이 모델은 알파벳 시퀀스를 역순으로 배열하는 Seq2Seq 모델임을 상기하며 [그림11] 그림을 보자.

<그림 시작>

그림11

[그림11: 알파벳 시퀀스 어텐션 매커니즘 - 인코더]

<그림 끝>

[그림11]는 [3 4 5]가 `inputs`으로 들어갈 때 인코더에서 일어나는 연산 과정을 그림으로 나타낸 것이다. 인코더 부분은 간단하다. 우선 `inputs`를 임베딩하여 `embedded`가 됐다. (##각주2 임베딩한 값을 3.0 4.0 5.0 등으로 체웠는데, 이는 이해를 쉽게 하기 위해 친숙한 숫자로 표현한 것이다. 실제로는 랜덤한 값으로 임베딩된다.) `embedded`는 `hidden`과 함께 GRU에 입력된다. GRU는 `output`과 `hidden`을 리턴한다. 여기에서 `hidden`은 [3 4 5]라는 시퀀스를 하나의 벡터로 나타낸 것이다. 여기에서 `hidden`만으로는 `inputs` 시퀀스를 모두 표현하기 힘들기 때문에 디코더에서 `output`을 이용해서 어탠션을 적용한다.

<그림 시작>

그림12

[그림12: 알파벳 시퀀스 어텐션 매커니즘 - 디코더:어텐션]

<그림 끝>

인코더에서 [3 4 5]를 `inputs`으로 넣었을 때 기대하는 디코더의 `output`은 [5 4 3]이다. 디코더에서 어텐션이 적용되는 방법이 [그림12]에 표현돼 있다. Seq2Seq 모델의 디코더는 [5 4 3]의 값을 하나씩 타임스탭마다 넣어준다. [5] 하나가 [그림12]의 `inputs`이다. `inputs`을 임베딩하여 `embedded`를 만들고 그것을 인코더의 `hidden`과 이어붙인다. 그렇게 이은 벡터를 `attn`을 통해서 행렬곱을 해준다. (##각주3 행렬곱의 의미는 선형대수학에서의 projection이다. 즉 다른 차원의 점으로 이동시키는 것이다.) 여기에 softmax를 취하면 `attn\_weight`을 얻을 수 있다. `attn\_weights`은 softmax를 취한 값이기 때문에 총 합이 1이고 길이는 `MAX\_LENGTH`이다. 이 값을 인코더의 `encoder\_outputs`에 곱해주면 인코더 `inputs` [3 4 5]에 가중치를 부여하는 셈이고 그것이 `attn\_applied`이다. `attn\_applied`와 `embedded`를 이어붙여보자. 이 두 값을 이어 붙인다는 것은 인코더의 입력 값과 디코더의 입력 값을 이어 붙인 셈이다. 이 값을 `attn\_combine`을 이용해 행렬 곱을 해서 relu 연산을 한 다음에 GRU 셀에 입력으로 넣는다. 핵심은 인코더와 디코더의 입력 값을 합쳐서 GRU에 넣는다는 것이고 인코더의 입력은 어텐션 가중치가 적용된 값이다.

이제 모델링하는 부분을 요약해보자. 우선 인코더를 만든다. 인코더의 입력인 `inputs`은 (batch\_size,max\_length) 형태의 벡터이다. 인코더의 출력은 `outputs`과 `hidden`인데, `outputs`은 (batch\_size,max\_length,hidden\_size)이며 `hidden`은 (batch\_size,1,hidden\_size)이다. [그림13]을 보자.

<그림 시작>

그림13

[그림13: 인코더 요약]

<그림 끝>

그 다음에 디코더를 만드는데 디코더는 인코더처럼 하나의 시퀀스를 통채로 처리하지 않고 하나 하나(auto-regressive) 시퀀스 길이만큼 반복하여 처리한다. 디코더가 입력으로 받아들이는 `inputs`은 (batch\_size, 1)과 앞서 계산했던 인코더의 출력인 `outputs`와 `hidden`이다. 이 과정을 디코더의 시퀀스 길이만큼 반복하게 된다. 정리하면 [그림14]와 같다.

<그림 시작>

그림14

[그림14: 디코더 요약]

<그림 끝>

인코더와 디코더의 입력과 출력을 하나로 정리하면 [그림15]와 같다.

<그림 시작>

그림15

[그림15: 인코더/디코더 요약]

<그림 끝>

<팁 시작>

팁: 모델링을 하기 전에 전체적인 입력과 출력의 형태를 정리한 후에 코딩을 하면 훨씬 간단하고 정리된 코드를 작성할 수 있다.

<팁 끝>

### 모델링 학습하기

`trainIters`함수를 이용해서 Seq2Seq 모델을 학습할 수 있다. `trainIters`내에는 `train`함수를 사용하고 있고 SGD 옵티마이져와 NLLLoss를 이용한다. [코드7]을 보자.

<코드 시작>

코드7

def train(input\_tensor, target\_tensor, encoder, decoder, encoder\_optimizer, decoder\_optimizer, criterion, max\_length=MAX\_LENGTH, with\_attention=True):

batch\_size = input\_tensor.size(0)

encoder\_hidden = encoder.initHidden(batch\_size)

encoder\_optimizer.zero\_grad()

decoder\_optimizer.zero\_grad()

input\_length = input\_tensor.size(1)

target\_length = target\_tensor.size(1)

encoder\_outputs = torch.zeros(max\_length, encoder.hidden\_size, device=device)

loss = 0

encoder\_outputs, encoder\_hidden = encoder(input\_tensor, encoder\_hidden)

decoder\_input = torch.tensor([bos]\*batch\_size, device=device)

decoder\_input = decoder\_input.unsqueeze(-1) # (B,1)

decoder\_hidden = encoder\_hidden

use\_teacher\_forcing = True if np.random.random() < teacher\_forcing\_ratio else False

for di in range(target\_length):

decoder\_output, decoder\_hidden, decoder\_attention = decoder(

decoder\_input, # (B,1)

decoder\_hidden, # (1,B,H)

encoder\_outputs # (B,M,H)

)

decoder\_output = decoder\_output.squeeze(1)

loss += criterion(decoder\_output, target\_tensor[:,di])

if use\_teacher\_forcing:

decoder\_input = target\_tensor[:,di].unsqueeze(-1) # (B,1)

else:

topv, topi = decoder\_output.topk(1)

decoder\_input = topi # (B,1)

loss.backward()

encoder\_optimizer.step()

decoder\_optimizer.step()

return loss.item() / target\_length

def trainIters(encoder, decoder, n\_iters, print\_every=1000, plot\_every=100, learning\_rate=0.01, with\_attention=True):

start = time.time()

plot\_losses = []

print\_loss\_total = 0 # print\_every 마다 초기화

plot\_loss\_total = 0 # plot\_every 마다 초기화

encoder\_optimizer = optim.SGD(encoder.parameters(), lr=learning\_rate)

decoder\_optimizer = optim.SGD(decoder.parameters(), lr=learning\_rate)

criterion = nn.NLLLoss()

for iter, batch in enumerate(train\_dataloader):

input\_tensor, target\_tensor, length\_tensor = batch

input\_tensor = input\_tensor.to(device)

target\_tensor = target\_tensor.to(device)

length\_tensor = length\_tensor.to(device)

loss = train(input\_tensor, target\_tensor, encoder,

decoder, encoder\_optimizer, decoder\_optimizer, criterion, with\_attention=with\_attention)

print\_loss\_total += loss

plot\_loss\_total += loss

if (iter+1) % print\_every == 0:

print\_loss\_avg = print\_loss\_total / print\_every

print\_loss\_total = 0

print('%s (%d %d%%) %.4f' % (timeSince(start, (iter+1) / n\_iters),

(iter+1), (iter+1) / n\_iters \* 100, print\_loss\_avg))

if (iter+1) % plot\_every == 0:

plot\_loss\_avg = plot\_loss\_total / plot\_every

plot\_losses.append(plot\_loss\_avg)

plot\_loss\_total = 0

showPlot(plot\_losses)

<코드 끝>

[코드7]에 대한 자세한 설명은 생략하려고 한다. 레포지토리에 있는 코드와 주피터 노트북을 통해서 스스로 학습해보도록 하자. [코드8]에서 모델을 선언하고 trainIters 함수로 모델을 학습해보자.

<코드 시작>

코드8

hidden\_size = 256

encoder1 = EncoderRNN(26+3, hidden\_size).to(device)

decoder1 = AttnDecoderRNN(hidden\_size, 26+3, dropout\_p=0.1).to(device)

for \_ in range(15):

trainIters(encoder1, decoder1, 75000, print\_every=30, with\_attention=True)

0m 0s (- 38m 29s) (30 0%) 3.0195

0m 1s (- 37m 7s) (60 0%) 2.8927

0m 2s (- 36m 36s) (90 0%) 2.7512

0m 3s (- 36m 23s) (120 0%) 2.6641

0m 4s (- 36m 14s) (150 0%) 2.4978

0m 5s (- 36m 7s) (180 0%) 2.3319

0m 0s (- 35m 46s) (30 0%) 2.0461

0m 1s (- 35m 45s) (60 0%) 1.9150

0m 2s (- 35m 45s) (90 0%) 1.7883

0m 3s (- 35m 44s) (120 0%) 1.6298

0m 4s (- 35m 45s) (150 0%) 1.6249

0m 5s (- 35m 45s) (180 0%) 1.5884

0m 0s (- 35m 50s) (30 0%) 1.4247

0m 1s (- 35m 52s) (60 0%) 1.3634

0m 2s (- 36m 24s) (90 0%) 1.3265

0m 3s (- 36m 18s) (120 0%) 1.2368

<코드 끝>

학습을 한 다음에는 학습한 모델을 검증해봐야 한다. [코드9]의 evaluate 함수를 통해서 검증해볼 수 있다.

<코드 시작>

코드9

def evaluate(encoder, decoder, input\_tensor, max\_length=MAX\_LENGTH):

with torch.no\_grad():

batch\_size = input\_tensor.size(0)

encoder\_hidden = encoder.initHidden(batch\_size)

#print(input\_tensor.shape, encoder\_hidden.shape)

encoder\_outputs, encoder\_hidden = encoder(input\_tensor, encoder\_hidden)

decoder\_input = torch.tensor([bos]\*batch\_size, device=device)

decoder\_input = decoder\_input.unsqueeze(-1) # (B,1)

decoder\_hidden = encoder\_hidden

decoded\_words = []

decoder\_attentions = torch.zeros(max\_length, max\_length)

for di in range(max\_length):

#print('\*\* ', decoder\_input.shape, decoder\_hidden.shape, encoder\_outputs.shape)

decoder\_output, decoder\_hidden, decoder\_attention = decoder(

decoder\_input, # (B,1)

decoder\_hidden, # (1,B,H)

encoder\_outputs # (B,M,H)

)

decoder\_output = decoder\_output.squeeze(1)

decoder\_attentions[:, di] = decoder\_attention

topv, topi = decoder\_output.topk(1)

decoder\_input = topi # (B,1)

if topi.item() == eos:

decoded\_words.append('</s>')

break

else:

decoded\_words.append(i2a[topi.item()])

return decoded\_words, decoder\_attentions[:di + 1]

for d in valid\_dataloader:

input\_tensor, target\_tensor, length\_tensor = d

input\_string = list(map(lambda i: i2a[i], input\_tensor.numpy()[0]))

output\_string = list(map(lambda i: i2a[i], target\_tensor.numpy()[0]))

input\_string = np.array(input\_string)

output\_string = np.array(output\_string)

print('input: {}'.format(input\_string))

print('output: {}'.format(output\_string))

input\_tensor = input\_tensor.to(device)

target\_tensor = target\_tensor.to(device)

length\_tensor = length\_tensor.to(device)

pred, attn\_weight = evaluate(encoder1, decoder1, input\_tensor)

pred = np.array(pred)

print('pred: {}'.format(pred))

print('-------------')

input: ['u' 'r' 'a' 'k' 't' 'r' 'g' 's' 'u' 'i' '</s>' '<pad>' '<pad>' '<pad>'

'<pad>']

output: ['i' 'u' 's' 'g' 'r' 't' 'k' 'a' 'r' 'u' '</s>' '<pad>' '<pad>' '<pad>'

'<pad>']

pred: ['i' 'u' 's' 'g' 'r' 't' 'k' 'a' 'r' 'u' '</s>']

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

input: ['v' 'b' 'z' 'p' 'v' 'n' 's' 'k' 'o' 'o' '</s>' '<pad>' '<pad>' '<pad>'

'<pad>']

output: ['o' 'o' 'k' 's' 'n' 'v' 'p' 'z' 'b' 'v' '</s>' '<pad>' '<pad>' '<pad>'

'<pad>']

pred: ['o' 'o' 'o' 'k' 's' 'n' 'v' 'p' 'z' 'b' 'v' '</s>']

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

input: ['w' 'g' 'l' 'e' 'm' 'h' 'o' 'q' 'c' 'r' '</s>' '<pad>' '<pad>' '<pad>'

'<pad>']

output: ['r' 'c' 'q' 'o' 'h' 'm' 'e' 'l' 'g' 'w' '</s>' '<pad>' '<pad>' '<pad>'

'<pad>']

pred: ['r' 'c' 'q' 'o' 'h' 'm' 'e' 'l' 'g' 'w' '</s>']

<코드 끝>

[코드9]의 결과를 보면 input, output, pred를 보여주고 있다. 모델에 input을 넣어서 pred를 생성해냈다. input을 역순으로 잘 생성한 것을 볼 수 있다. [코드9]에서 사용한 evaluate 함수를 잘 보자. 학습할 때와 다르게 teacher forcing이 들어가지 않고 오로지 모델이 한 글자씩 출력한 것을 연결한 것이다. teacher forcing은 모델 학습을 효과적으로 해주기 위함이므로 모델을 검증할 때는 사용하지 않는다.

마지막으로 어텐션을 출력해서 시각화해주는 코드를 작성해보자.

<코드 시작>

코드10

def showAttention(input\_sentence, output\_words, attentions):

# colorbar로 그림 설정

fig = plt.figure()

ax = fig.add\_subplot(111)

cax = ax.matshow(attentions.numpy(), cmap='bone')

fig.colorbar(cax)

# 축 설정

ax.set\_xticklabels([''] + input\_sentence.split(' ') +

['</s>'], rotation=90)

ax.set\_yticklabels([''] + output\_words)

# 매 틱마다 라벨 보여주기

ax.xaxis.set\_major\_locator(ticker.MultipleLocator(1))

ax.yaxis.set\_major\_locator(ticker.MultipleLocator(1))

plt.show()

def evaluateAndShowAttention():

input\_tensor = make\_alphabet\_tensor()

output\_words, attentions = evaluate(

encoder1, decoder1, input\_tensor)

input\_sentence = input\_tensor[0].tolist()

input\_sentence = list(map(lambda x: i2a[x], input\_sentence))

print('original input =', input\_sentence)

input\_length = input\_sentence.index('<pad>')

input\_sentence = ' '.join(input\_sentence[:input\_length])

print('input =', input\_sentence)

print('output =', ' '.join(output\_words))

showAttention(input\_sentence, output\_words, attentions)

evaluateAndShowAttention()

<코드 끝>

[코드10]을 실행시키면 [그림16]와 같은 결과를 얻는다.

<그림 시작>

그림16

역방향 어텐션 그림

<그림 끝>

[그림16]은 [코드10]을 실행시켰을 때 계산되는 어텐션을 시각화한 결과이다. 가로가 input 방향이고 세로가 output 방향이다. input의 제일 첫 문자인 k를 생성할 때 output의 제일 뒷 문자인 </s> 부분에 가장 높은 어텐션 값을 갖는 것을 알 수 있다.

이번 장에서는 어탠션 네트워크에 대해서 자세하게 알아봤다. 어탠션 네트워크의 개념과 필요한 이유에 대해서 설명했고, 어탠션 네트워크가 어떤 구조를 띄고 있고 실제 구현에서 어떻게 사용되는지 예제 소스코드를 통해서도 알아봤다. 어탠션 네트워크를 자세하게 다룬 이유는 이 책의 핵심인 3장과 4장을 이해하기 위해 가장 우선적으로 필요한 개념이기 때문이다. 3장에서는 2장에서 공부한 어탠션 네트워크 개념을 이용해서 트랜스포머를 공부해보자.