

프로젝트 최종 보고서

시나리오 챗봇과 생성 기반 챗봇 비교

국민대학교 소프트웨어융합대학원
인공지능전공

최현규

- **챗봇이란?**

챗봇이란 사람과 기계가 대화를 할 수 있도록 설계된 소프트웨어이다. 문자나 음성으로 대화할 수 있으며 사용자의 의도(Intent)를 잘 파악하여 사용자에게 유익한 정보(Entity)를 전달할 수 있는 기계를 말한다.

- **챗봇 기본 용어**

- 1) 자연어(Natural Language)

- 자연어란 사람이 일상 생활에서 사용하는 언어를 말한다.

- 2) 자연어처리(Natural Language Processing)

- 자연어처리란 자연어의 의미를 분석하여 컴퓨터가 처리하게끔 하는 일련의 과정을 말한다. 자연어처리는 음성 인식, 번역, 감성 분석, 텍스트 분류, 챗봇과 같은 곳에서 사용되는 분야이다. 컴퓨터가 인간의 언어를 알아들을 수 있게 만드는 학문분야로써, 인공지능의 하위 분야로 일반적인 인공지능을 만들려던 1960년대 연구 시도가 실패한 후 인간의 언어를 분석하고 해석하여 처리하는 인공지능이 세분화되면서 생긴 분야이기도 하다.

- 3) 자연어이해(Natural Language Understanding)

- 자연어이해란 자연어 표현을 기계가 이해할 수 있는 다른 표현으로 변환하는 것을 말한다. 형태소 분석이나 구문 분석과 같은 자연어 처리와 혼용해서 사용되는 경우가 많으나, 자연어이해가 더 큰 개념이다. 단순히 단어나 문장의 형태를 기계가 인식하도록 하는 것이 아니라 의미를 인식하도록 하는 것을 의미한다. 문장 의도 분류, 언어간 번역, 문장 생성, 자연어 질문에 대한 답변 등이 있다.

- 4) 의도(Intent)

- 인간이 기계에게 전달하는 목적(입력의 주제)을 뜻한다. 기계와 대화를 하는 것은 목적이 있는 경우가 대부분이기 때문에 인간에 의해 설계된 챗봇은 입력에 대해 유의미한 출력을 사용자에게 전달해야한다. 따라서 퀄리티가 높은 챗봇을 만들기 위해서는 설계 부분에서 Intent를 최대한 좋은 방향으로 설계해야한다.

- 5) 개체명(Entity)

- 인간이 기계에게 전달하는 입력의 키워드를 말한다. 챗봇이 인간과 대화를 할 수 있으려면 다음과 같은 작업을 수행할 수 있어야 한다. 가. 사용자의 입력을 특정 패턴으로 추출 및 가공할 수 있다. 나. 사용자의 입력을 해석할 수 있어야 한다. 다. 사용자의 입력에 대해 적절한 출력과 부가적인 정보를 제공할 수 있다.

- **주제 선정 이유 – 챗봇이 사회에 미치는 영향**

자연어이해 분야가 상용화되기 이전에는 음성을 입력으로 사용하여 특정 앱을 디자인하려고 했을 때, 출력을 얻기 위해 수 많은 입력 샘플이 필요했다. 자연어이해가 상용화된 이후에는 하나의 출력에 대해서 수 많은 입력들이 본질적으로 동일한 것임을 알게 할 수 있는 것이 가능해졌기 때문에 베이스라인과 학습에 사용될 데이터만 충분하다면 퀄리티가 우수한 챗봇을 제작할 수 있다.

챗봇을 제작한다고 해서 챗봇만을 바라보며 연구를 하는 것은 아니다. 위에서 언급했다시피 챗봇을 제작한다는 것은 기본적으로 자연어처리와 자연어이해를 동시에 다루는 것이고 근본적으로 Text-To-Text의 형태보단 Text-To-Speech(TTS), Speech-To-Text(STT) 그리고 TTS와 STT가 접목된(영화에서나 볼 수 있었던 AI의 형태) 기술을 바라본다는 의미이다.

따라서 자연어 처리와 자연어이해는 기계와 소통할 수 있는 수단에 있어서 연구를 하는 것이 중요한 부분이고, 결과적으로 인공지능이 인간에게 서비스될 때 어떠한 형태의 인공지능이 가장 편리한 수단이 될 수 있는지가 더욱 중요한 부분이 된다.

현재 프로젝트에서는 간단한 챗봇 구현을 통해, 비슷한 네트워크를 기준으로 서로 다른 챗봇이 어떤 곳에 사용되어야 하는지 머신러닝(딥러닝)을 통해 직접 결과를 도출하고 비교해본다.

- **애플리케이션 제작 방안**

모델 구성 – 시나리오형 챗봇, 생성 기반 챗봇

Sequence to Sequence Network based on Attention

하나의 시퀀스를 다른 시퀀스로 변경하는 두 개의 RNN(Encoder, Decoder)이 함께 동작하는 네트워크를 구성하여 기존의 머신러닝보다 우수한 결과를 도출할 수 있는 신경망이다.

Sequence to Sequence(Seq2Seq)란 2014년에 발표된 논문에서 소개된 모델이다. 일반적인 딥러닝 모델에서는 모델에 사용되는 다수의 입력의 길이가 동일해야하고 임베딩을 통해 동일한 차원으로 맞추어야한다. 하지만 Seq2Seq 신경망에서는 Decoder의 입출력 부분에서 문장의 시작과 끝을 알리는 토큰(Start-Of-Sentence: SOS, End-Of-Sentence: EOS)을 사용함으로써 길이가 다른 다수의 입력을 사용할 수 있게 했다. 경우에 따라서 Padding을 사용할 수 있기 때문에 SOS와 EOS를 정의할 때 PAD 토큰을 별도로 정의하기도 한다. Seq2Seq는 기본적으로 두 개의 RNN을 사용한다. 하나는 Encoder, 다른 하나는 Decoder를 이루고 있다.

논문에서 인용한 예시는 다음과 같다.

‘LSTM(Long Term Short Term Memory – 장단기 메모리 LSTM)’은 딥러닝 분야에서 사용되는 Recurrent Neural Network(RNN)이다. 장기 의존성 문제를 해결할 수 있다는 점에서 직전 데이터뿐만 아니라, 조금 더 거시적으로 이전 데이터를 고려하여 이후의 데이터를 예측한다. 다만 메모리가 덮어쓰워질 가능성이 있고, 여전히 길이가 긴 시퀀스에 대해서 정보를 전부 기억할 수 없는 단점이 있다.

입력이 ‘ABC’ 일 때, 출력은 ‘WXYZ’ 이다. 기본적으로 지도학습을 지향하고 있기 때문에 데이터의 구성은 Input-Target 구조이다. 입력인 ‘ABC’가 처음 접하는 부분을 ‘Encoder’라고 부르고 Encoder를 이루고 있는 각 Sell은 ‘LSTM Sell’로 이루어져 있다. 일반적으로 RNN 계열의 Sell로 구성하지만 입력이 긴 시퀀스를 처리하기 위해서는 LSTM Sell을 사용한다. Encoder에 입력된 데이터는 모델이 요구하는 형태에 맞게 Encoding을 진행하여 각 Sell로 전달하고 Encoder의 마지막 Sell(Encoder의 마지막 Hidden state라고 말하기도 한다)에서 Decoder의 initial hidden state로 제공할 준비를 한다.

‘WXYZ’를 출력하는 부분을 ‘Decoder’라고 하는데 Decoder 역시 Encoder처럼 LSTM Sell로 이루어져 있다. Decoder의 입력은 사전에 정의된 시작을 알리는 토큰(Start-Of-Sentence: SOS)로 시작되고 SOS토큰과 SOS 다음에 위치한 Target Sentence가 사용된다. Decoder의 출력은 Decoding된 Sentence이고 SOS 토큰과 같이 사전에 정의된 종료를 알리는 토큰(End-Of-Sentence: EOS)로 종료된다.

- **애플리케이션 제작 방안**

논문에서 제시된 Decoder의 학습 방법은 Beam Search라는 방법이고 그 내용은 다음과 같다.

일반적으로 LSTM Sell에서 사용하는 argmax 방법은 각 Sell을 거칠 때마다 확률값을 최대로 갖는 단어를 출력값으로 학습하는 것이다. 이 방법은 LSTM Sell에서 학습 도중 잘못 예측한 데이터가 있다면 다음 데이터의 학습에도 지대한 영향을 미치기 때문에 학습 전체의 과정에 커다란 문제가 될 수 있는 여지가 크다. 이러한 문제를 해결하기 위해 Beam Search라는 방법을 인용했다.

각 Sell에서 N 가지의 가설을 두고 Sentence의 종료 부분에서 가설의 argmax를 통해 최종 출력을 결정한다. 학습 과정에서는 입력과 출력 데이터 Pair가 존재하기 때문에 Beam Search 방법을 사용할 수 있지만 예측과정에서는 입력만 존재하기 때문에 Beam Search 방법을 사용할 수 있다. LSTM의 단점을 보완하고 개선하기 위해 Attention Mechanism을 사용하여 Decoder가 입력 시퀀스의 특정 범위에 집중할 수 있도록 한다.

시나리오형 챗봇의 경우 추천시스템 정도를 제외하고는 딥러닝이 적용되지 않는다. 현재 프로젝트에서는 챗봇 자체의 구성과 제작 방법을 다룰 것이므로 시나리오형 챗봇에는 딥러닝을 적용하지 않았다. (데이터 또한 시나리오형 챗봇에는 적용되지 않았으며 별도로 사전을 구성하였다.)

• 생성 기반 챗봇 데이터 구성

Q	A	label
12시 땡!	하루가 또 가네요.	0
1지망 학교 떨어졌어	위로해 드립니다.	0
3박4일 놀러가고 싶다	여행은 언제나 좋죠.	0
3박4일 정도 놀러가고 싶다	여행은 언제나 좋죠.	0
PPL 심하네	눈살이 찌푸려지죠.	0
SD카드 망가졌어	다시 새로 사는 게 마음 편해요.	0
SD카드 안돼	다시 새로 사는 게 마음 편해요.	0
SNS 맞팔 왜 안하지ㅠㅠ	잘 모르고 있을 수도 있어요.	0
SNS 시간낭비인 거 아는데 매일 하는 중	시간을 정하고 해보세요.	0
SNS 시간낭비인데 자꾸 보게됨	시간을 정하고 해보세요.	0
SNS보면 나만 빠고 다 행복해보여	자랑하는 자리니까요.	0
가끔 궁금해	그 사람도 그럴 거예요.	0
가끔 뭐하는지 궁금해	그 사람도 그럴 거예요.	0
가끔은 혼자인게 좋다	혼자를 즐기세요.	0
가난한 자의 설움	돈은 다시 들어올 거예요.	0
가만 있어도 땀난다	땀을 식혀주세요.	0
가상화폐 쫄딱 망함	어서 잊고 새출발 하세요.	0
가스불 켜고 나갔어	빨리 집에 돌아가서 끄고 나오세요.	0
가스불 켜놓고 나온거 같아	빨리 집에 돌아가서 끄고 나오세요.	0
가스비 너무 많이 나왔다.	다음 달에는 더 절약해봐요.	0
가스비 비싼데 감기 걸리겠어	따뜻하게 사세요!	0
가스비 장난 아님	다음 달에는 더 절약해봐요.	0

- 해당 데이터는 https://github.com/songys/Chatbot_data 에서 다운로드 받을 수 있으며 Input-Target으로 이루어져 있다.
- 챗봇 트레이닝용 문답 11,876 페어로 구성되어 있으며, 일상 다반서 0, 이별(부정) 1, 사랑(긍정) 2로 레이블링 되어 있다.

- 시나리오 챗봇 구성

```
...  
module import  
...  
  
# !pip install PyKomoran  
from PyKomoran import *  
from collections import OrderedDict, Counter  
import pandas as pd  
import json
```

```
...  
komoran model setting  
...  
  
komoran = Komoran(model_path = DEFAULT_MODEL['FULL'])  
komoran.set_user_dic('./data/dic/user.dic')
```

- 시나리오 챗봇 구성을 위한 모듈을 임포트한다.

• 시나리오형 챗봇 구성

```
...  
module import  
...  
# !pip install PyKomoran  
from PyKomoran import *  
from collections import OrderedDict, Counter  
import pandas as pd  
import json
```

```
...  
komoran model setting  
...
```

```
komoran = Komoran(model_path = DEFAULT_MODEL['FULL'])  
komoran.set_user_dic('./data/dic/user.dic')
```



user - Windows 메모장

파일(F) 편집(E) 서식(O) 보기(V) 도움말(H)

```
# burger  
맥스파이시 상하이 NNP  
맥스파이시상하이 NNP  
상하이 스파이시 NNP  
상하이스파이시 NNP  
상하이 NNP  
빅맥 NNP  
맥치킨 NNP  
맥치킨 모짜렐라 NNP  
맥치킨모짜렐라 NNP  
더블 불고기 NNP  
더블불고기 NNP
```

user 속성

일반

보안

자세히

이전 버전



user

파일 형식:

텍스트 문서(.dic)

연결 프로그램:

메모장

변경(C)...

위치:

C:\Users#gusrb#workplace\AI\Chatbot\Cu:

크기:

6.53KB (6,693 바이트)

디스크 할당 크기:

8.00KB (8,192 바이트)

- PyKomoran 모델 경로를 입력한 한다.
- Komoran.set_user_dic(PATH + Filename)에서 사용자(또는 컨셉)에 맞게 사전을 구성한다. 텍스트 파일로 구성하되, 저장 시 확장자는 .dic로 저장한다.

<https://komorandocs.readthedocs.io/ko/latest/pykomoran/installation.html>

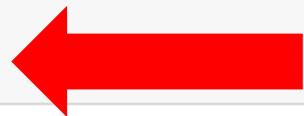
- 시나리오형 챗봇 구성

```
...
menu list loading
...
PATH = './data/dic/menu_list/'
burger = pd.read_csv(PATH + 'burger.txt')['burger'].tolist()
beverage = pd.read_csv(PATH + 'beverage.txt')['beverage'].tolist()
size = pd.read_csv(PATH + 'size.txt')['size'].tolist()
menu_type = pd.read_csv(PATH + 'menu_type.txt')['menu_type'].tolist()
dessert = pd.read_csv(PATH + 'dessert.txt')['dessert'].tolist()
icecream = pd.read_csv(PATH + 'icecream.txt')['icecream'].tolist()
morning = pd.read_csv(PATH + 'morning.txt')['morning'].tolist()

...
save as json file, intent list for customer in store
...
menu_list = OrderedDict()
menu_list['burger'] = burger
menu_list['beverage'] = beverage
menu_list['size'] = size
menu_list['menu_type'] = menu_type
menu_list['dessert'] = dessert
menu_list['icecream'] = icecream
menu_list['morning'] = morning

with open(PATH + 'menu_list.json', 'w', encoding = 'utf-8') as file:
    json.dump(menu_list, file, ensure_ascii = False, indent = '  ')
```

- 사용자 사전과는 별개로 사용자의 입력을 직접 조회하기 위한 사전을 별도로 구성한다.
- User.dic에 저장된 데이터와 동일해야하므로 User.dic에 있는 데이터를 목적에 맞게 분리하여 Json 형식의 파일로 저장한다.



- 시나리오형 챗봇 구성

```
...  
data loading  
...  
with open('./data/dic/menu_list/menu_list.json', 'r', encoding = 'utf-8') as file:  
    intent_dict = json.load(file)  
  
print(intent_dict.keys())  
  
dict_keys(['burger', 'beverage', 'size', 'menu_type', 'dessert', 'icecream', 'morning'])
```

- 저장한 Json 파일을 불러온다.

• 시나리오형 챗봇 구성

```
print(intent_dict.values())
```

dict_values(['맥스파이시 상하이', '맥스파이시상하이', '상하이 스파이시', '상하이스파이시', '상하이', '빅맥', '맥치킨', '맥치킨 모짜렐라', '맥치킨모짜렐라', '더블 불고기', '더블불고기', '에그 불고기', '에그불고기', '불고기', '슈슈', '슈비', '베이컨 토마토 디럭스', '베이컨토마토 디럭스', '베이컨 토마토디럭스', '베이컨토마토디럭스', '베토디', '더블 쿼터파운더 치즈', '더블쿼터파운더 치즈', '더블쿼터파운더치즈', '쿼터파운더 치즈', '쿼터파운더치즈', '치즈', '더블 치즈', '더블치즈', '햄', '일구오오', '1955', '더블 1955', '더블1955', '1955 해쉬브라운', '1955해쉬브라운'], ['코카-콜라', '코카콜라', '콜라', '사이다', '스프라이트', '환타', '코카-콜라 제로', '코카-콜라제로', '코카 콜라 제로', '코카콜라제로', '제로 코카-콜라', '제로 코카 콜라', '제로코카 콜라', '제로 코카콜라', '제로코카콜라', '제로콜라', '제로코크', '제로콕', '아이스 커피', '아이스커피', '뜨거운커피', '핫커피', '생수', '물', '우유', '밀크', '오렌지 주스', '오렌지주스', '오렌지 주스', '오렌지주스', '주스', '주스', '딸기 칠러', '딸기칠러', '배 칠러', '배칠러', '바닐라 셰이크', '바닐라셰이크', '바닐라셰이크', '딸기 셰이크', '딸기셰이크', '딸기셰이크', '초코 셰이크', '초코셰이크', '초코셰이크', '카페라떼', '카페라떼', '디카페인 카페라떼', '디카페인카페라떼', '디카페인 카페라떼', '디카페인카페라떼', '아이스 카페라떼', '아이스카페라떼', '아이스 카페라떼', '아이스카페라떼', '디카페인 아이스 카페라떼', '디카페인아이스 카페라떼', '디카페인아이스 카페라떼', '디카페인아이스카페라떼', '디카페인 아이스 카페라떼', '디카페인아이스 카페라떼', '디카페인아이스카페라떼', '아메리카노', '디카페인 아메리카노', '디카페인아메리카노', '디카페인아메리카노', '아이스 아메리카노', '아이스아메리카노', '아아', '디카페인 아이스 아메리카노', '디카페인아이스 아메리카노', '디카페인아이스 아메리카노', '디카페인 아이스 아메리카노', '디카페인아이스 아메리카노', '디카페인아이스 아메리카노', '디카페인아이스아메리카노', '카푸치노', '디카페인 카푸치노', '디카페인카푸치노', '디카페인 카푸치노', '디카페인카푸치노', '에스프레소', '디카페인 에스프레소', '디카페인에스프레소', '디카페인 에스프레소', '디카페인에스프레소', '프리미엄 로스트 원두커피', '프리미엄로스트 원두커피', '프리미엄로스트 원두커피'], ['미디움', '라지'], ['세트', '단품'], ['맥너겟 4조각', '맥너겟 4개', '맥너겟4조각', '맥너겟4개', '맥너겟 네 개', '맥너겟 네개', '맥너겟네개', '맥너겟 네 조각', '맥너겟 네조각', '맥너겟네조각', '맥너겟 6조각', '맥너겟 6개', '맥너겟6조각', '맥너겟6개', '맥너겟 여섯 개', '맥너겟 여섯개', '맥너겟여섯개', '맥너겟 여섯 조각', '맥너겟 여섯조각', '맥너겟 10조각', '맥너겟 10개', '맥너겟10조각', '맥너겟10개', '맥너겟 열 개', '맥너겟 열개', '맥너겟열개', '맥너겟 열 조각', '맥너겟 열조각', '맥너겟열조각'], ['바나나 오레오 맥플러리', '바나나오레오 맥플러리', '바나나오레오맥플러리', '바나나 오레오', '바나나오레오', '바나나 맥플러리', '바나나맥플러리', '딸기 오레오 맥플러리', '딸기오레오 맥플러리', '딸기오레오맥플러리', '딸기 오레오', '딸기오레오', '딸기 맥플러리', '딸기맥플러리', '초코 오레오 맥플러리', '초코오레오 맥플러리', '초코오레오맥플러리', '초코 오레오', '초코오레오', '초코 맥플러리', '초코맥플러리', '오레오 맥플러리', '오레오맥플러리', '아이스크림콘', '아이스크림 콘', '콘', '초코콘', '초코 콘', '초코 선데이 아이스크림', '초코선데이 아이스크림', '초코선데이아이스크림', '초코 선데이', '초코선데이', '카라멜 선데이 아이스크림', '카라멜선데이 아이스크림', '카라멜선데이아이스크림', '카라멜 선데이', '카라멜선데이', '딸기 선데이 아이스크림', '딸기선데이 아이스크림', '딸기선데이아이스크림', '딸기 선데이', '딸기선데이', '선데이', '오레오 애플파이', '애플파이', '치킨 치즈 머핀', '치킨치즈머핀', '에그 맥머핀', '에그맥머핀', '베이컨 에그 맥머핀', '베이컨에그 맥머핀', '베이컨에그맥머핀', '소시지 에그 맥머핀', '소시지에그 맥머핀', '소세지 에그 맥머핀', '소세지에그 맥머핀', '베이컨 토마토 머핀', '베이컨토마토 머핀', '소시지 맥머핀', '소시지맥머핀', '소세지 맥머핀', '소세지맥머핀', '디럭스 블랙퍼스트', '디럭스블랙퍼스트', '디럭스 블랙퍼스트', '디럭스블랙퍼스트', '핫케익 3조각', '핫케익3조각', '핫케익 세 조각', '핫케익 세조각', '핫케익 3개', '핫케익3개', '핫케

- 시나리오형 챗봇 구성

```
# preprocessing number to string
def num2str(order_dict):
    for index, value in enumerate(order_dict):
        if order_dict[index] == '0':
            order_dict[index] = '영'
        elif order_dict[index] == '1':
            order_dict[index] = '일'
        elif order_dict[index] == '2':
            order_dict[index] = '이'
        elif order_dict[index] == '3':
            order_dict[index] = '삼'
        elif order_dict[index] == '4':
            order_dict[index] = '사'
        elif order_dict[index] == '5':
            order_dict[index] = '오'
        elif order_dict[index] == '6':
            order_dict[index] = '육'
        elif order_dict[index] == '7':
            order_dict[index] = '칠'
        elif order_dict[index] == '8':
            order_dict[index] = '팔'
        elif order_dict[index] == '9':
            order_dict[index] = '구'

    return order_dict
```

- 사용자의 특정 문구에 숫자가 입력될 경우
글자로 변경하는 함수를 정의한다.

사용자 사전에 별도로 입력하여 출력하려고 시도하였으나 잘
진행되지 않았음

- 시나리오형 챗봇 구성

```
# extract intent from customer
def extract_intent(input_sentence):
    ...
    str2dict with customer
    ...
    order_dict = {index: value for index, value in enumerate([i for i in input_sentence])}
    tkn_order = "".join(list(num2str(order_dict).values()))
    customer_order = {'CUSTOMER_ORDER': komoran.nouns(tkn_order)}

    for index, samples in enumerate(customer_order.items()):
        key, value = samples
        for index, word in enumerate(value):
            if '안녕' in word:
                del value[index]

    customer_intent_count = Counter(value)

    return customer_order, customer_intent_count
```

- 사용자의 입력을 전처리한다.
- 입력을 토큰화할 때 인사말이 제거될 수 있도록 한다.

- 시나리오형 챗봇 구성

```
# matching
def matching_menu(order, intent_dict):
    customer_dict = {}
    for index, items in enumerate(intent_dict.items()): # v: list
        category, menues = items
        menu = [value for value in menues if value in order]
        customer_dict.update({category: menu})

    return customer_dict
```

- 사용자의 입력을 토큰화한 뒤, 사전에 입력해둔 단어들과 매칭한다.

- 시나리오형 챗봇 구성

```
# response
def response(customer_dict):
    # 기본 햄버거 주문
    if len(customer_dict['burger']) != 0 and len(customer_dict['morning']) == 0:
        print("요청하신 {}, {}, {}-{} {} {} 주문이 완료되었습니다.".format(
            customer_dict['burger'],
            customer_dict['beverage'],
            customer_dict['size'],
            customer_dict['menu_type'],
            customer_dict['dessert'],
            customer_dict['icecream']
        ))

    # 기본 모닝 주문
    elif len(customer_dict['burger']) == 0 and len(customer_dict['morning']) != 0:
        print("요청하신 {}, {}, {}-{} {} {} 주문이 완료되었습니다.".format(
            customer_dict['morning'],
            customer_dict['beverage'],
            customer_dict['size'],
            customer_dict['menu_type'],
            customer_dict['dessert'],
            customer_dict['icecream']
        ))

    # 햄버거, 모닝 외 주문
    elif len(customer_dict['burger']) == 0 and len(customer_dict['morning']) == 0:
        print("요청하신 {} {} {} 주문이 완료되었습니다.".format(
            customer_dict['beverage'],
            customer_dict['dessert'],
            customer_dict['icecream']
        ))

    # 햄버거, 모닝 동시 주문 시
    elif len(customer_dict['burger']) != 0 and len(customer_dict['morning']) != 0:
        print("손님, 요청하신 {} 제품과 {} 제품은 동시에 주문이 어렵습니다. 다른 제품을 주문해주세요."
            .format(customer_dict['burger'], customer_dict['morning']))
```

- 사용자에게 출력될 문구를 형식에 맞게 미리 제작한다.

- 시나리오형 챗봇 구성

```
# order menu
def chatting(input_sentence, intent_dict):
    customer_intents, counts = extract_intent(input_sentence)
    # dict2list with store
    store_list = [intent for categories, menus in intent_dict.items() for intent in menus]
    # CUSTOMER's order list
    order = [value for value in store_list if value in customer_intents['CUSTOMER_ORDER']]
    # Matching between CUSTOMER's order and menus in the store
    customer_dict = matching_menu(order, intent_dict)
    # Answer
    response(customer_dict)

    return customer_dict
```

- 사전 정의된 extract_intent(), matching_menu(), response() 함수를 통해 실제 결과를 도출하는 함수를 정의한다.

- 시나리오형 챗봇 구성

```
# chat with bot
def chat(intent_dict):
    input_sentence = ""
    while(1):
        try:
            ...
            주문 접수 여부
            ...

            input_sentence = input('주문하시겠습니까? > ')
            if input_sentence == '아니오' or input_sentence == '아니요':
                print("감사합니다. 안녕히 가십시오.")
                break

            ...
            주문 접수
            ...

            customer_dict = chatting(input_sentence, intent_dict)

            ...
            주문 종료
            ...

            check_sentence = input('주문을 이어서 하시겠습니까? > ')
            if check_sentence == '아니오' or check_sentence == '아니요':
                print("주문이 완료되었습니다.")
                break
            elif check_sentence == '예' or check_sentence == '네': continue
        except:
            print("Error")
```

- 사용자와 챗봇의 연속적인 대화를 위한 함수를 정의한다.

- 시나리오형 챗봇 구성

```
...
Result 1
...
input_sentence_1 = "안녕하세요, 상하이버거 미디움세트, 빅맥버거 라지세트에 음료수는 아이스크피, 콜라로 주세요."
input_sentence_2 = "상하이버거 라지세트하나에 빅맥단품으로 주세요. 아 그리고 상하이 버거에는 콜라로 주세요"
input_sentence_3 = "빅맥 단품 하나 주세요."
input_sentence_4 = "더블1955버거 단품 하나 주세요."
input_sentence_5 = "슈슈버거 세트에 음료수는 환타로 주세요."
input_sentence_6 = "소세지 에그 맥머핀 세트하나에 음료수는 배 칠러로 주세요. 아 그리고 아이스크림콘도 같이 주세요."
input_sentence_7 = "핫케익3개 주시면 감사하겠습니다." # 수량 출력 수정해야됨

input_sentence_list = [
    "안녕하세요, 상하이버거 미디움세트, 빅맥버거 라지세트에 음료수는 아이스크피, 콜라로 주세요.",
    "상하이버거 라지세트하나에 빅맥단품으로 주세요. 아 그리고 상하이 버거에는 콜라로 주세요",
    "빅맥 단품 하나 주세요.",
    "더블1955버거 단품 하나 주세요.", # 숫자 출력 수정해야됨
    "슈슈버거 세트에 음료수는 환타로 주세요.",
    "소세지 에그 맥머핀 세트하나에 음료수는 배 칠러로 주세요. 아 그리고 아이스크림콘도 같이 주세요.",
    "핫케익3개 주시면 감사하겠습니다.", # 숫자 출력 수정해야됨
    "베이컨 토마토 머핀 단품 하나에 빅맥 세트로 주세요. 음료수는 우유로 부탁드립니다."
]
```

- 챗봇에 입력할 데이터를 별도로 정의한다.

- 시나리오형 챗봇 결과

```
for i in range(len(input_sentence_list)):
    print(chatting(input_sentence_list[i], intent_dict), sep = '\n')
    print()
```

요청하신 ['상하이', '빅맥'], ['콜라', '아이스커피'], ['미디움', '라지']-['세트'] [] [] 주문이 완료되었습니다.

{'burger': ['상하이', '빅맥'], 'beverage': ['콜라', '아이스커피'], 'size': ['미디움', '라지'], 'menu_type': ['세트'], 'dessert': [], 'icecream': [], 'morning': []}

요청하신 ['상하이', '빅맥'], ['콜라'], ['라지']-['세트', '단품'] [] [] 주문이 완료되었습니다.

{'burger': ['상하이', '빅맥'], 'beverage': ['콜라'], 'size': ['라지'], 'menu_type': ['세트', '단품'], 'dessert': [], 'icecream': [], 'morning': []}

요청하신 ['빅맥'], [], []-['단품'] [] [] 주문이 완료되었습니다.

{'burger': ['빅맥'], 'beverage': [], 'size': [], 'menu_type': ['단품'], 'dessert': [], 'icecream': [], 'morning': []}

요청하신 ['일구오오'], [], []-['단품'] [] [] 주문이 완료되었습니다.

{'burger': ['일구오오'], 'beverage': [], 'size': [], 'menu_type': ['단품'], 'dessert': [], 'icecream': [], 'morning': []}

요청하신 ['슈슈'], ['환타'], []-['세트'] [] [] 주문이 완료되었습니다.

{'burger': ['슈슈'], 'beverage': ['환타'], 'size': [], 'menu_type': ['세트'], 'dessert': [], 'icecream': [], 'morning': []}

요청하신 ['소세지 에그 맥머핀'], ['배 칠러'], []-['세트'] [] ['아이스크림콘'] 주문이 완료되었습니다.

{'burger': [], 'beverage': ['배 칠러'], 'size': [], 'menu_type': ['세트'], 'dessert': [], 'icecream': ['아이스크림콘'], 'morning': ['소세지 에그 맥머핀']}

요청하신 ['핫케익'], [], []-[] [] [] 주문이 완료되었습니다.

{'burger': [], 'beverage': [], 'size': [], 'menu_type': [], 'dessert': [], 'icecream': [], 'morning': ['핫케익']}

손님, 요청하신 ['빅맥'] 제품과 ['베이컨 토마토 머핀'] 제품은 동시에 주문이 어렵습니다. 다른 제품을 주문해주세요.

{'burger': ['빅맥'], 'beverage': ['우유'], 'size': [], 'menu_type': ['세트', '단품'], 'dessert': [], 'icecream': [], 'morning': ['베이컨 토마토 머핀']}

- 예시로 작성한 스크립트를 출력한다.

- 시나리오형 챗봇 결과

```
chat(intent_dict)
```

주문하시겠습니까? > 안녕하세요, 음...빅맥세트하나에 음료수는 배 칠러로 주세요.
요청하신 ['빅맥'], ['배 칠러'], []-['세트'] [] [] 주문이 완료되었습니다.
주문을 이어서 하시겠습니까? > 네
주문하시겠습니까? > 1955버거 단품도 추가해주세요
요청하신 ['일구오오'], [], []-['단품'] [] [] 주문이 완료되었습니다.
주문을 이어서 하시겠습니까? > 네
주문하시겠습니까? > 오레오 맥플러리도 하나 부탁드립니다.
요청하신 [] [] ['오레오 맥플러리'] 주문이 완료되었습니다.
주문을 이어서 하시겠습니까? > 아니요
주문이 완료되었습니다.

```
chat(intent_dict)
```

주문하시겠습니까? > 네 안녕하세요! 맥스파이시 상하이버거 세트 하나 부탁드립니다.
요청하신 ['맥스파이시 상하이'], [], []-['세트'] [] [] 주문이 완료되었습니다.
주문을 이어서 하시겠습니까? > 네
주문하시겠습니까? > 음료수는 아이스 아메리카노로 주세요.
요청하신 ['아이스 아메리카노'] [] [] 주문이 완료되었습니다.
주문을 이어서 하시겠습니까? > 아니요
주문이 완료되었습니다.

- 챗봇과 대화해본다.

- **생성 기반 챗봇 구성** - 현재 스크립트는 Pytorch Tutorial을 기반으로 작성되었음

```
import csv
import random
import re
import os
import unicodedata
import itertools
import warnings

import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
```

- 생성 기반 챗봇을 구성하기 위해 필요한 모듈을 임포트한다.
- 기본적으로 Pytorch로 코드를 구성하며 tokenizer는 konlpy 형식의 tokenizer보단 python 기본 내장 함수인 split() 함수를 사용한다. 데이터의 양이 방대해지기 때문에 konlpy의 함수를 사용한다면 학습 시간이 오래 걸릴뿐더러 전처리 시간도 오래 걸리기 때문이다.

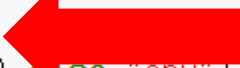
- 생성 기반 챗봇 구성

```
warnings.filterwarnings("ignore", category=UserWarning)
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

- GPU를 사용하기 위해 device라는 변수에 저장한다.

- 생성 기반 챗봇 구성

```
warnings.filterwarnings("ignore", category=UserWarning)  
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```



- 함수를 종종 사용하다보면 경고 메시지가 나타나는 경우가 있기 때문에 경고를 무시하는 명령어를 추가한다.

• 생성 기반 챗봇 구성

```
corpus_name = "normal chatbot data"
corpus = os.path.join("data", corpus_name)
textfilename = "chatbot_data.txt"

def printLines(file, n=10):
    with open(file, 'r', encoding = 'utf-8') as datafile:
        lines = datafile.readlines()
        for line in lines[:n]:
            print(line)

printLines(os.path.join(corpus, textfilename))
datafile = os.path.join(corpus, textfilename)
```

12시 땡! 하루가 또 가네요.

1지망 학교 떨어졌어 위로해 드립니다.

3박4일 놀러가고 싶다 여행은 언제나 좋죠.

3박4일 정도 놀러가고 싶다 여행은 언제나 좋죠.

PPL 심하네 눈살이 찌푸려지죠.

SD카드 망가졌어 다시 새로 사는 게 마음 편해요.

SD카드 안돼 다시 새로 사는 게 마음 편해요.

- Corpus_name은 학습에 사용될 데이터의 이름이다. 또한 데이터가 저장될 폴더의 이름이기도 하다.
- printLines() 함수는 로딩한 데이터의 출력을 목적으로 하는 함수이다.

- 생성 기반 챗봇 구성

```
# 기본 단어 토큰 값
PAD_token = 0 # 짧은 문장을 채움(패딩, padding) 때 사용할 제로 토큰
SOS_token = 1 # 문장의 시작(SOS, Start Of Sentence)을 나타내는 토큰
EOS_token = 2 # 문장의 끝(EOS, End Of Sentence)을 나타는 토큰
```

- 앞서 설명한 학습에 사용될 토큰을 정의한다.

• 생성 기반 챗봇 구성

```
class Voc:
    def __init__(self, name):
        self.name = name
        self.trimmed = False
        self.word2index = {}
        self.word2count = {}
        self.index2word = {PAD_token: "PAD", SOS_token: "SOS", EOS_token: "EOS"}
        self.num_words = 3 # SOS, EOS, PAD: 3개

    def addSentence(self, sentence):
        for word in sentence.split(' '):
            self.addWord(word)

    def addWord(self, word):
        if word not in self.word2index:
            self.word2index[word] = self.num_words
            self.word2count[word] = 1
            self.index2word[self.num_words] = word
            self.num_words += 1
        else:
            self.word2count[word] += 1

# 등장 횟수가 기준 이하인 단어를 정리합니다
    def trim(self, min_count):
        if self.trimmed:
            return
        self.trimmed = True

        keep_words = []

        for k, v in self.word2count.items():
            if v >= min_count:
                keep_words.append(k)

        print('keep_words {} / {} = {:.4f}'.format(
            len(keep_words), len(self.word2index), len(keep_words) / len(self.word2index)
        ))

# 사전을 다시 초기화합니다
        self.word2index = {}
        self.word2count = {}
        self.index2word = {PAD_token: "PAD", SOS_token: "SOS", EOS_token: "EOS"}
        self.num_words = 3 # 기본 토큰을 세 곳

        for word in keep_words:
            self.addWord(word)
```

- 질의/응답 문장 페어를 저장할 단어 사전을 정의한다.
- 해당 tutorial에 따르면, 일련의 seq들은 단어들이며 이들을 이산 공간 상의 수치로 자연스럽게 대응하기 어렵기 때문에 데이터 셋 안에 들어 있는 단어를 인덱스 값으로 변환하는 매핑을 별도로 만들어야 한다고 한다.
- 따라서 Voc라는 클래스를 만들어 단어에서 인덱스로의 매핑, 인덱스에서 단어로의 역 매핑, 각 단어의 등장 횟수, 전체 단어 수 등을 관리한다.
- 현재 클래스에는 단어 사전에 새로운 단어를 추가하는 메서드(addWord), 문장에 등장하는 단어를 추가하는 메서드(addSentence), 자주 등장하지 않는 단어를 정리하는 메서드(trim)이 들어 있다.

• 생성 기반 챗봇 구성

```
MAX_LENGTH = 15 # 고려할 문장의 최대 길이

# 유니코드 문자열을 아스키로 변환합니다
# https://stackoverflow.com/a/518232/2809427 참고
def unicodeToAscii(s):
    return ''.join(
        c for c in unicodedata.normalize('NFD', s)
        if unicodedata.category(c) != 'Mn'
    )

# 소문자로 만들고, 공백을 넣고, 알파벳 외의 글자를 제거합니다
def normalizeString(s):
    hangul = re.compile('[^ㄱ-ㅣ가-힣^☆;^a-zA-Z.!?:0-9]+')
    result = hangul.sub('', s)
    return result

# 질의/응답 쌍을 위해서 voc 객체를 반환합니다
def readVocs(datafile, corpus_name):
    print("Reading lines...")
    # 파일을 읽고, 포개어 lines에 저장합니다
    lines = open(datafile, encoding='utf-8').read().strip().split('\n')
    # 각 줄을 포개어 pairs에 저장하고 정규화합니다
    pairs = [[normalizeString(s) for s in l.split('\t')] for l in lines]
    voc = Voc(corpus_name)
    return voc, pairs

# 문장의 쌍 'p'에 포함된 두 문장이 모두 MAX_LENGTH라는 기준보다 짧은지를 반환합니다
def filterPair(p):
    # EOS 토큰을 위해 입력 시퀀스의 마지막 단어를 보존해야 합니다
    return len(p[0].split(' ')) < MAX_LENGTH and len(p[1].split(' ')) < MAX_LENGTH

# 조건식 filterPair에 따라 pairs를 필터링합니다
def filterPairs(pairs):
    return [pair for pair in pairs if filterPair(pair)]
```

- 단어사전과 질의/응답 문장 페어를 재구성하기 위해 unicodeToAscii() 함수를 이용하여 유니코드 문자열을 아스키로 변환한다.
- normalizeString() 함수에 정규표현식을 적용하여 불필요한 문자를 제거한다.
- 학습의 편의를 위해서 filterPair() 함수와 filterPairs() 함수를 이용해 전처리될 문장의 최대 길이를 설정한다.

• 생성 기반 챗봇 구성

```
# 앞에서 정의한 함수를 이용하여 만든 voc 객체와 리스트 pairs를 반환합니다
def loadPrepareData(corpus, corpus_name, datafile, save_dir):
    print("Start preparing training data ...")
    voc, pairs = readVocs(datafile, corpus_name)
    print("Read {!s} sentence pairs".format(len(pairs)))
    pairs = filterPairs(pairs)
    print("Trimmed to {!s} sentence pairs".format(len(pairs)))
    print("Counting words...")
    for pair in pairs:
        voc.addSentence(pair[0])
        voc.addSentence(pair[1])
    print("Counted words:", voc.num_words)
    return voc, pairs

# voc와 pairs를 읽고 재구성합니다
save_dir = os.path.join("data", "save")
voc, pairs = loadPrepareData(corpus, corpus_name, datafile, save_dir)
# 검증을 위해 pairs의 일부 내용을 출력해 봅니다
print("#npairs:")
for pair in pairs[:10]:

    print(pair)
```

```
Start preparing training data ...
Reading lines...
Read 11823 sentence pairs
Trimmed to 11812 sentence pairs
Counting words...
Counted words: 21720
```

```
pairs:
['12시 땡!', '하루가 또 가네요.']
['1지망 학교 떨어졌어', '위로해 드립니다.']
['3박4일 놀러가고 싶다', '여행은 언제나 좋죠.']
['3박4일 정도 놀러가고 싶다', '여행은 언제나 좋죠.']
['PPL 심하네', '눈살이 찌푸려지죠.']
['SD카드 망가졌어', '다시 새로 사는 게 마음 편해요.']
['SD카드 안돼', '다시 새로 사는 게 마음 편해요.']
['SNS 맞팔 왜 안하지ㅠㅠ', '잘 모르고 있을 수도 있어요.']
['SNS 시간낭비인 거 아는데 매일 하는 중', '시간을 정하고 해보세요.']
['SNS 시간낭비인데 자꾸 보게됨', '시간을 정하고 해보세요.']
```

- 앞서 정의한 함수들을 이용하여 voc 객체와 pairs 리스트를 반환한다.
- Voc 객체와 pairs 리스트를 저장하고 재구성한다.

• 생성 기반 챗봇 구성

```
MIN_COUNT = 0 # 제외할 단어의 기준이 되는 등장 횟수

def trimRareWords(voc, pairs, MIN_COUNT):
    # MIN_COUNT 미만으로 사용된 단어는 voc에서 제외합니다
    voc.trim(MIN_COUNT)
    # 제외할 단어가 포함된 경우를 pairs에서도 제외합니다
    keep_pairs = []
    for pair in pairs:
        input_sentence = pair[0]
        output_sentence = pair[1]
        keep_input = True
        keep_output = True
        # 입력 문장을 검사합니다
        for word in input_sentence.split(' '):
            if word not in voc.word2index:
                keep_input = False
                break
        # 출력 문장을 검사합니다
        for word in output_sentence.split(' '):
            if word not in voc.word2index:
                keep_output = False
                break

        # 인출된 문장에 제외하기로 한 단어를 포함하지 않는 경우만을 남겨둡니다
        if keep_input and keep_output:
            keep_pairs.append(pair)

    print("Trimmed from {} pairs to {}, {:.4f} of total".format(
        len(pairs), len(keep_pairs), len(keep_pairs) / len(pairs)))
    return keep_pairs

# voc와 pairs를 정돈합니다
pairs = trimRareWords(voc, pairs, MIN_COUNT)
```

```
keep_words 21717 / 21717 = 1.0000
Trimmed from 11812 pairs to 11812, 1.0000 of total
```

- trimRareWords() 함수를 이용하여 학습 단계가 빨리 수렴할 수 있도록 빈도수가 작은 단어를 제거한다.
- 해당 함수에는 앞서 정의한 Voc.trim 함수를 이용하여 MIN_COUNT 라는 기준 이하의 단어를 제거한다.

- 생성 기반 챗봇 구성

```
def indexesFromSentence(voc, sentence):
    return [voc.word2index[word] for word in sentence.split(' ')] + [EOS_token]

def zeroPadding(l, fillvalue=PAD_token):
    return list(itertools.zip_longest(*l, fillvalue=fillvalue))

def binaryMatrix(l, value=PAD_token):
    m = []
    for i, seq in enumerate(l):
        m.append([])
        for token in seq:
            if token == PAD_token:
                m[i].append(0)
            else:
                m[i].append(1)
    return m
```

- Tutorial에 따르면, 데이터 타입을 torch.tensor로 변환해야하기 때문에 전처리된 데이터를 모델이 맞는 형태로 변경해야한다.
- 학습 속도나 GPU 병렬 처리 용량을 향상하려면 데이터를 배치로 분할하여 학습한다. 같은 배치 안에서 크기가 다른 문장을 처리하기 위해서는 배치용 입력 텐서의 모양을 (max_length, batch_size)로 맞춰야 한다. 이때 max_length보다 짧은 문장에 대해서는 EOS 토큰 뒤에 제로 토큰을 덧붙인다.

• 생성 기반 챗봇 구성

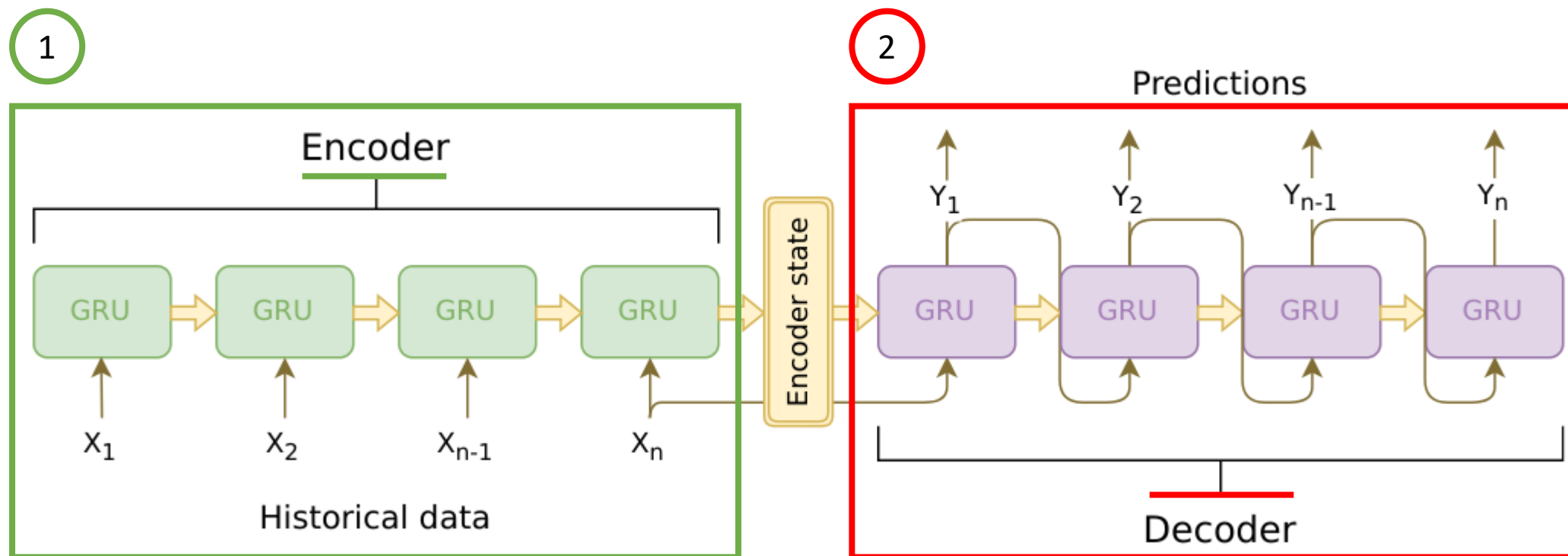
```
# 입력 시퀀스 텐서에 패딩한 결과와 lengths를 반환합니다
def inputVar(l, voc):
    indexes_batch = [indexesFromSentence(voc, sentence) for sentence in l]
    lengths = torch.tensor([len(indexes) for indexes in indexes_batch])
    padList = zeroPadding(indexes_batch)
    padVar = torch.LongTensor(padList)
    return padVar, lengths

# 패딩한 목표 시퀀스 텐서, 패딩 마스크, 그리고 최대 목표 길이를 반환합니다
def outputVar(l, voc):
    indexes_batch = [indexesFromSentence(voc, sentence) for sentence in l]
    max_target_len = max([len(indexes) for indexes in indexes_batch])
    padList = zeroPadding(indexes_batch)
    mask = binaryMatrix(padList)
    mask = torch.ByteTensor(mask)
    padVar = torch.LongTensor(padList)
    return padVar, mask, max_target_len

# 입력 배치를 이루는 쌍에 대한 모든 아이템을 반환합니다
def batch2TrainData(voc, pair_batch):
    pair_batch.sort(key=lambda x: len(x[0].split(" ")), reverse=True)
    input_batch, output_batch = [], []
    for pair in pair_batch:
        input_batch.append(pair[0])
        output_batch.append(pair[1])
    inp, lengths = inputVar(input_batch, voc)
    output, mask, max_target_len = outputVar(output_batch, voc)
    return inp, lengths, output, mask, max_target_len
```

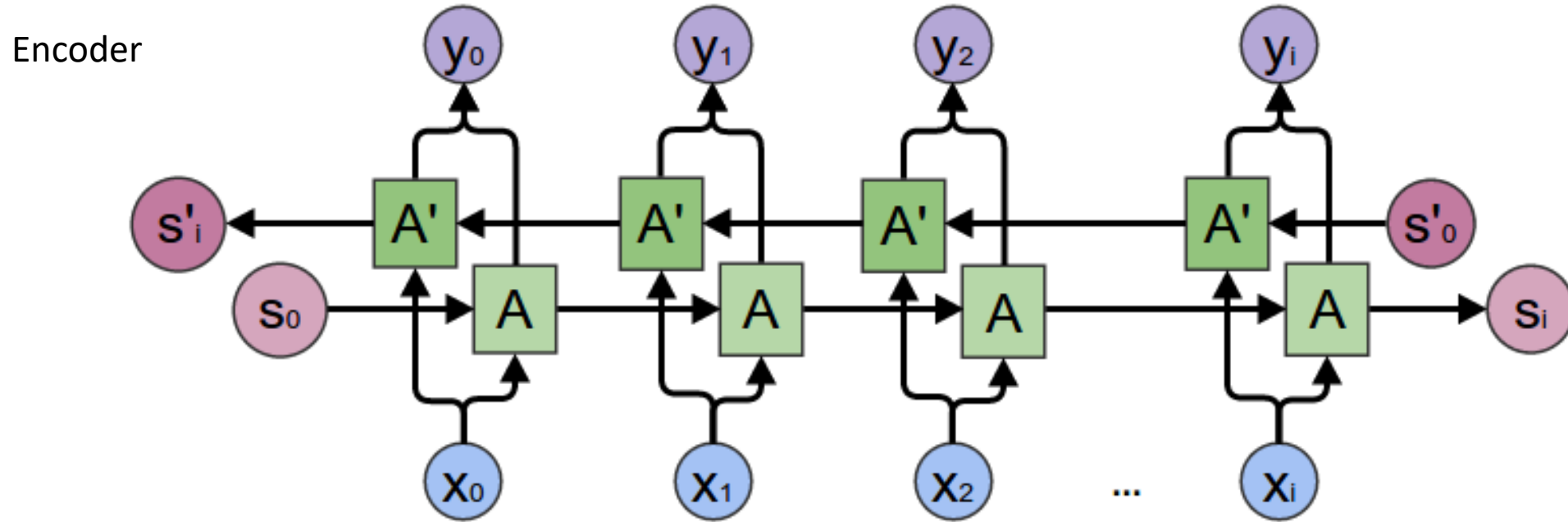
- inputVar() 함수를 통해 입력 시퀀스 텐서에 패딩한 결과와 Length를 반환한다.
- outputVar() 함수를 통해 패딩한 목표 시퀀스 텐서, 패딩 마스크, 그리고 최대 목표 길이를 반환한다.
- batch2TrainData() 함수를 통해 입력 배치를 이루는 쌍에 대한 모든 아이템들을 반환한다.

- 생성 기반 챗봇 구성



- Seq2Seq: 독립된 두 개의 순환 신경망을 같이 사용하여 원하는 결과를 얻기 위한 모델
- 하나의 순환 신경망은 Encoder로, 가변 길이 입력 시퀀스를 고정된 길이의 문맥 벡터(context vector)로 Encoding한다.
- 다른 하나의 순환 신경망은 Decoder로, 단어 하나와 문맥 벡터를 입력으로 받고 다음 시퀀스가 어떤 시퀀스일지 예측한다.

- 생성 기반 챗봇 구성



- Encoder는 입력 시퀀스를 토큰 단위로 한번에 하나씩 계산한다. 그리고 각 단계마다 출력 벡터와 은닉 상태 벡터를 반환한다.
- 은닉 상태 벡터는 다음 단계를 진행할 때 같이 사용되며, 출력 벡터는 차례대로 기록된다.

• 생성 기반 챗봇 구성

```
class EncoderRNN(nn.Module):
    def __init__(self, hidden_size, embedding, num_layers=1, dropout=0.2):
        super(EncoderRNN, self).__init__()
        self.num_layers = num_layers
        self.hidden_size = hidden_size
        self.embedding = embedding

        # GRU를 초기화합니다. input_size와 hidden_size 파라미터는 둘 다 'hidden_size'로
        # 둡니다. 이는 우리 입력의 크기가 hidden_size 만큼의 피처를 갖는 단어 임베딩이기
        # 때문입니다.
        self.gru = nn.GRU(
            input_size = self.hidden_size,
            hidden_size = self.hidden_size,
            num_layers = self.num_layers,
            dropout=(0 if num_layers == 1 else dropout),
            bidirectional=True)

    def forward(self, input_seq, input_lengths, hidden=None):
        # 단어 인덱스를 임베딩으로 변환합니다
        embedded = self.embedding(input_seq)
        # RNN 모듈을 위한 패딩된 배치 시퀀스를 패킹합니다
        packed = nn.utils.rnn.pack_padded_sequence(embedded, input_lengths, enforce_sorted=False)
        # GRU로 포워드 패스를 수행합니다
        outputs, hidden = self.gru(input_size = packed, hidden_size = hidden)
        # 패딩을 언패킹합니다
        outputs, _ = nn.utils.rnn.pad_packed_sequence(outputs)
        # 양방향 GRU의 출력을 합산합니다
        outputs = outputs[:, :, :self.hidden_size] + outputs[:, :, self.hidden_size:]
        # 출력과 마지막 은닉 상태를 반환합니다
        return outputs, hidden
```

- Seq2Seq 모델을 정의한다.
- Seq2Seq 모델의 목표는 가변 길이 시퀀스를 입력으로 받고, 크기가 고정된 모델을 이용하여, 가변 길이 시퀀스를 출력으로 반환하는 것이다.
- Encoder는 가변 길이 입력 시퀀스를 고정된 길이의 문맥 벡터(context vector)로 Encoding한다. 이론상 문맥 벡터(RNN의 마지막 hidden state)는 입력에 대한 의미론적 정보를 담고 있을 것이다.
- 입력
Input_seq: (max_length, batch_size)

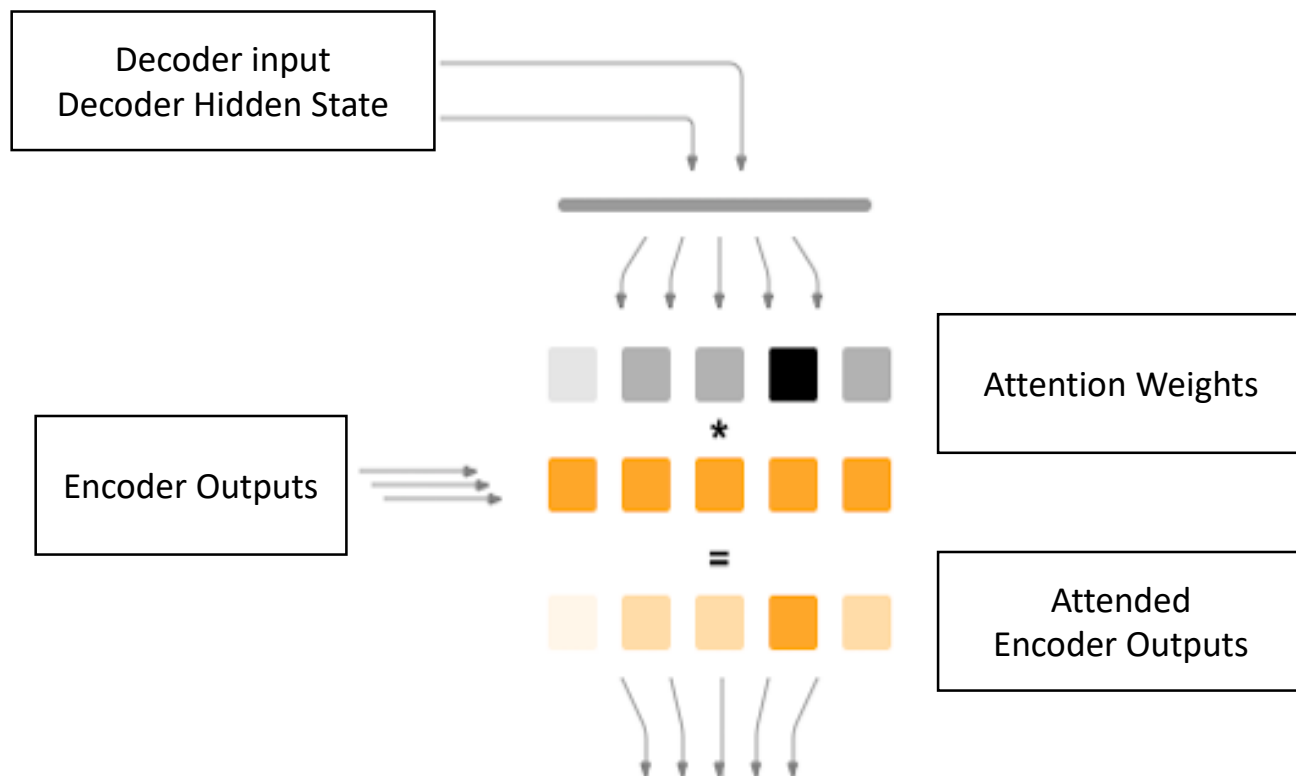
input_lengths: (batch_size)

hidden: (num_layers * num_directions, batch_size, hidden_size)
- 출력
outputs: (max_length, batch_size, hidden_size)

hidden: (num_layers * num_directions, batch_size, hidden_size)

- 생성 기반 챗봇 구성

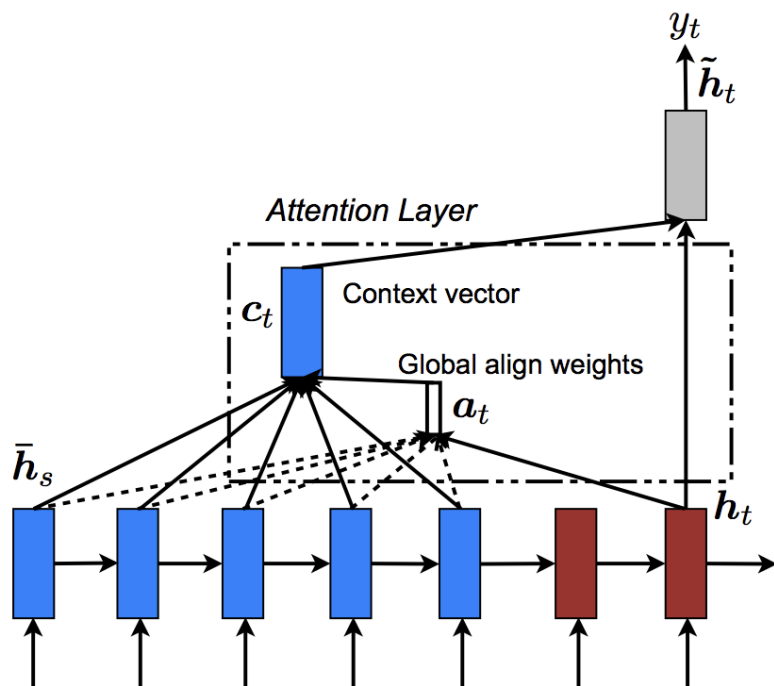
Local Attention - Bahdanau



- 입력 시퀀스의 의미를 인코딩할 때 문맥 벡터에만 전적으로 의존한다면, 그 과정 중에 정보 손실이 일어날 가능성이 높기 때문에 디코더의 기능이 크게 저하될 수 있다. 이를 해결하기 위해 고안된 방안을 Attention Mechanism라고 한다.
- Attention은 Decoder의 input, hidden state와 Encoder의 Outputs을 바탕으로 계산된다.
- 출력되는 Attention Weights는 Input sequence와 동일한 모양을 가지므로 Encoder의 Output과 곱할 수 있음에 따라 얻어지게 되는 Weight의 합은 Encoder의 Output에서 어느 부분에 집중해야 할지 알려준다.

- 생성 기반 챗봇 구성

Global Attention - Luong



$$\text{score}(\mathbf{h}_t, \bar{\mathbf{h}}_s) = \begin{cases} \mathbf{h}_t^\top \bar{\mathbf{h}}_s & \text{dot} \\ \mathbf{h}_t^\top \mathbf{W}_a \bar{\mathbf{h}}_s & \text{general} \\ \mathbf{v}_a^\top \tanh(\mathbf{W}_a [\mathbf{h}_t; \bar{\mathbf{h}}_s]) & \text{concat} \end{cases}$$

- 기존의 Attention Mechanism보다 발전된 Global-Attention이 발표되었는데, 기존과의 차이점은 Encoder의 모든 hidden state를 고려한다는 점이다. Local-Attention 방식이 현재 시점에 대한 Encoder의 hidden state만을 고려한다는 점과 다른 부분이다.
- Global-Attention과 Local-Attention과의 또 다른 차이점은 Attention에 대한 가중치, 혹은 에너지를 계산할 때 현재 시점에 대한 Decoder의 hidden state만을 사용한다는 점이다. Local-Attention에서는 Attention을 계산할 때 이전 단계 상태에 대한 hidden state만을 사용한다.
- Global-Attention에서는 Encoder의 출력과 Decoder의 출력에 대한 Attention 에너지를 계산하는 방법을 제공하며 이를 score function이라고 부른다.
- h_t : 계산하고자하는 Decoder의 현재 상태
 h_s : Encoder의 모든 상태

• 생성 기반 챗봇 구성

```
# Luong 어텐션 레이어
class Attn(nn.Module):
    def __init__(self, method, hidden_size):
        super(Attn, self).__init__()
        self.method = method
        if self.method not in ['dot', 'general', 'concat']:
            raise ValueError(self.method, "is not an appropriate attention method.")
        self.hidden_size = hidden_size
        if self.method == 'general':
            self.attn = nn.Linear(self.hidden_size, hidden_size)
        elif self.method == 'concat':
            self.attn = nn.Linear(self.hidden_size * 2, hidden_size)
            self.v = nn.Parameter(torch.FloatTensor(hidden_size))

    def dot_score(self, hidden, encoder_output):
        return torch.sum(hidden * encoder_output, dim=2)

    def general_score(self, hidden, encoder_output):
        energy = self.attn(encoder_output)
        return torch.sum(hidden * energy, dim=2)

    def concat_score(self, hidden, encoder_output):
        energy = self.attn(
            torch.cat((hidden.expand(encoder_output.size(0), -1, -1), encoder_output), 2)
        ).tanh()
        return torch.sum(self.v * energy, dim=2)

    def forward(self, hidden, encoder_outputs):
        # Attention 가중치(에너지)를 제안된 방법에 따라 계산합니다
        if self.method == 'general':
            attn_energies = self.general_score(hidden, encoder_outputs)
        elif self.method == 'concat':
            attn_energies = self.concat_score(hidden, encoder_outputs)
        elif self.method == 'dot':
            attn_energies = self.dot_score(hidden, encoder_outputs)

        # max_length와 batch_size의 차원을 뒤집습니다
        attn_energies = attn_energies.t()

        # 정규화된 softmax 확률 점수를 반환합니다 (차원을 늘려서)
        return F.softmax(attn_energies, dim=1).unsqueeze(1)
```

- Global-Attention Mechanism을 구현하면 다음과 같다.
Attention Layer를 Attn이라는 독립적인 nn.Module로 구현했고, 이 모듈의 출력은 (batch_size, 1, max_length)인 정규화된 Softmax **Weight** tensor이다.
- 전체적인 계산 과정은 다음과 같다.
 1. 현재 입력 단어에 대한 임베딩을 구한다
 2. 무방향 GRU로 Forward pass를 수행한다
 3. (2)에서 구한 현재의 GRU 출력을 바탕으로 Attn weight를 구한다
 4. Encoder output에 Attn weight를 곱하여 새로운 문법 벡터를 구한다
 5. 가중치 문법 벡터와 GRU 출력을 결합한다
 6. softmax 없이 다음 단어를 예측한다
 7. output과 hidden state를 반환한다

• 생성 기반 챗봇 구성

```
class LuongAttnDecoderRNN(nn.Module):
    def __init__(self, attn_model, embedding, hidden_size, output_size, num_layers=1, dropout=0.1):
        super(LuongAttnDecoderRNN, self).__init__()

        # 참조를 보존해 둡니다
        self.attn_model = attn_model
        self.hidden_size = hidden_size
        self.output_size = output_size
        self.n_layers = num_layers
        self.dropout = dropout
        # 레이어를 정의합니다
        self.embedding = embedding
        self.embedding_dropout = nn.Dropout(dropout)
        self.gru = nn.GRU(
            input_size = self.hidden_size,
            hidden_size = self.hidden_size,
            num_layers = self.num_layers,
            dropout=(0 if self.num_layers == 1 else dropout))
        self.concat = nn.Linear(self.hidden_size * 2, self.hidden_size)
        self.out = nn.Linear(self.hidden_size, self.output_size)
        self.attn = Attn(attn_model, hidden_size)

    def forward(self, input_step, last_hidden, encoder_outputs):
        # 주의: 한 단위 시퀀스에 대해 한 단계(단어)만을 수행합니다
        # 현재의 입력 단어에 대한 임베딩을 구합니다
        embedded = self.embedding(input_step)
        embedded = self.embedding_dropout(embedded)
        # 무방향 GRU로 포워드 패스를 수행합니다
        rnn_output, hidden = self.gru(input_size = embedded, hidden_size = last_hidden)
        # 현재의 GRU 출력을 바탕으로 어텐션 가중치를 계산합니다
        attn_weights = self.attn(rnn_output, encoder_outputs)
        # 인코더 출력에 어텐션을 곱하여 새로운 "가중치 합" 문맥 벡터를 구합니다
        context = attn_weights.bmm(encoder_outputs.transpose(0, 1))
        # Luong의 논문에서 나온 식 5를 이용하여 가중치 문맥 벡터와 GRU 출력을 결합합니다
        rnn_output = rnn_output.squeeze(0)
        context = context.squeeze(1)
        concat_input = torch.cat((rnn_output, context), 1)
        concat_output = torch.tanh(self.concat(concat_input))
        # Luong의 논문에서 나온 식 6을 이용하여 다음 단어를 예측합니다
        output = self.out(concat_output)
        output = F.softmax(output, dim=-1)
        # 출력과 마지막 은닉 상태를 반환합니다
        return output, hidden
```

- Attention 서브 모듈을 정의하고 나면 실제 Decoder 모델을 구현할 수 있다.
- Decoder에 대해 매 시간마다 배치를 하나씩 입력하게 되면 임베딩된 tensor와 GRU Output이 모두 (1, batch_size, hidden_size)가 된다.
- 입력
Input_step : (1, batch_size)
last_hidden : (num_layers * num_directions, batch_size, hidden_size)
encoder_outputs : (max_length, batch_size, hidden_size)
- 출력
output : (batch_size, vo.num_words)
hidden : (num_layers * num_directions, batch_size, hidden_size)

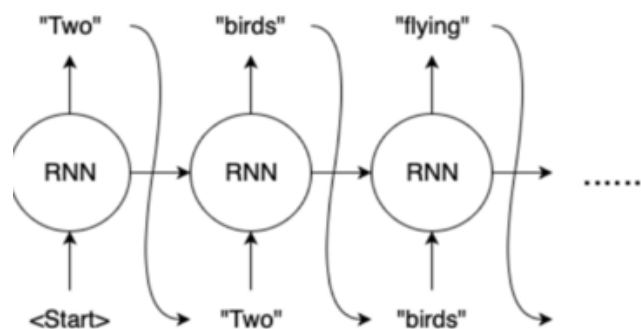
- 생성 기반 챗봇 구성

```
def maskNLLLoss(inp, target, mask):  
    nTotal = mask.sum()  
    crossEntropy = -torch.log(torch.gather(inp, 1, target.view(-1, 1)).squeeze(1))  
    loss = crossEntropy.masked_select(mask).mean()  
    loss = loss.to(device)  
    return loss, nTotal.item()
```

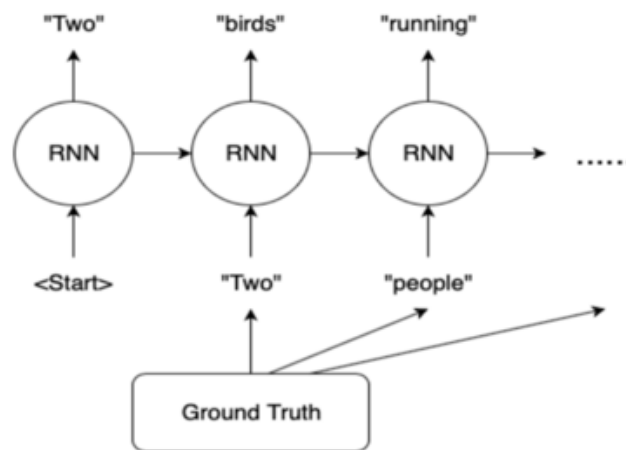
- 이 과정에서는 패딩된 시퀀스 배치를 다루기 때문에 손실을 계산하는 과정에서 모든 Pad Tensor를 고려할 수는 없다. 따라서 maskNLLLoss() 함수를 정의하여 Decoder의 Output Tensor, Target Tensor, Bi-masked Tensor를 바탕으로 Loss를 계산한다.

- 생성 기반 챗봇 구성

Teacher Forcing



Without Teacher Forcing



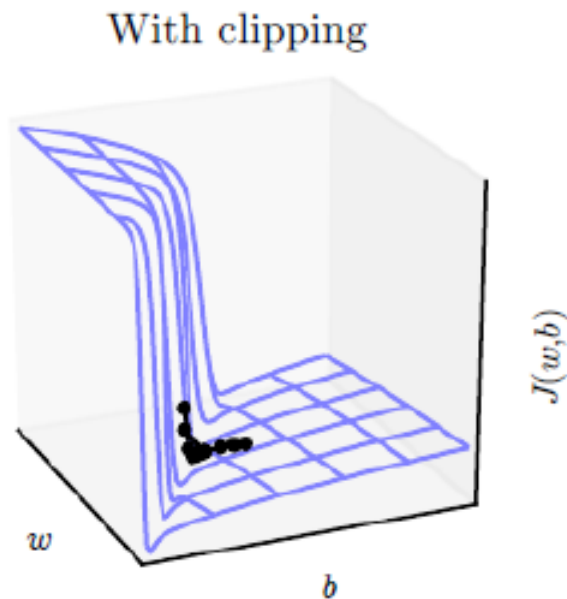
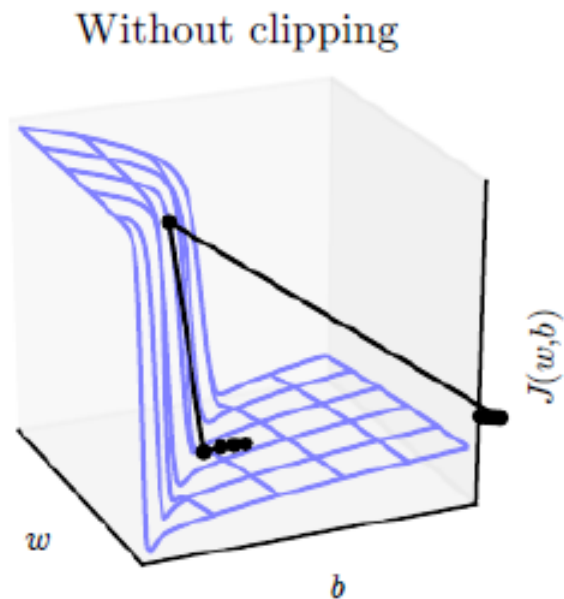
With Teacher Forcing

- 학습 과정에서 수렴이 잘 되게 적용되는 방법 중 하나로 Teacher Forcing 기법이 있다.
- Teacher Forcing이란, `teacher_forcing_ratio`로 정의된 확률에 따라서 Decoder의 이번 단계 예측값 대신에 현재의 목표 단어 Target word(Ground Truth)를 Decoder의 다음 입력값으로 활용하는 것이다.
- Attention 기법을 이용하여 예측 성능을 향상시켰다 하더라도, 학습 과정에서 잘못된 추론으로 인해 잘못된 예측으로 이어질 수 있기 때문에 Seq2Seq 모델에서 Teacher Forcing 기법을 사용한다.
- Teacher Forcing 기법은 추론 과정에서 모델 자체가 불안정하게 할 수 있기 때문에 `teacher_forcing_ratio` 값을 적절히 설정해야한다(노출 편향 문제).
- 노출 편향 문제가 생각만큼 큰 영향을 미치지 않는다는 연구 결과가 있다.

<https://arxiv.org/abs/1905.10617>

- 생성 기반 챗봇 구성

Gradient Clipping



- 학습 과정에서 수렴이 잘 되게 적용되는 방법 중 다른 하나는 Gradient Clipping 기법이다.
- 일반적으로 널리 사용되는 방법 중 하나이며, 핵심은 Gradient를 clipping하거나 임계값을 둬으로써, Gradient가 지수함수적으로 증가하거나 Overflow를 일으키는 경우를 막고 Cost Function의 급격한 경사를 피하는 방법이다.
- 전체적인 학습 과정은 다음과 같다.
 1. 전체 입력 배치에 대해 Encoder로 Forward Pass를 수행
 2. Decoder의 입력을 SOS_token으로, hidden state를 Encoder의 마지막 hidden state로 초기화
 3. 입력 배치 시퀀스를 한 번에 하나씩 Decoder로 Forward Pass를 수행
 4. Teacher Forcing을 사용하는 경우, Decoder의 다음 입력을 현재의 target으로 설정하고 Teacher Forcing을 사용하지 않는 경우에는 Decoder의 다음 입력을 현재 Decoder의 출력으로 설정
 5. Loss를 계산하고 기록
 6. BackPropagation 수행
 7. Gradient Clipping 수행
 8. Encoder, Decoder의 Parameter 갱신

• 생성 기반 챗봇 구성

1

```
def train(input_variable, lengths, target_variable, mask, max_target_len, encoder, decoder, embedding,
          encoder_optimizer, decoder_optimizer, batch_size, clip, max_length=MAX_LENGTH):

    # 새로 그라디언트
    encoder_optimizer.zero_grad()
    decoder_optimizer.zero_grad()

    # device 옵션을 설정합니다
    input_variable = input_variable.to(device)
    lengths = lengths.to(device)
    target_variable = target_variable.to(device)
    mask = mask.to(device)

    # 변수를 초기화합니다
    loss = 0
    print_losses = []
    n_totals = 0

    # 인코더로 포워드 패스를 수행합니다
    encoder_outputs, encoder_hidden = encoder(input_variable, lengths)

    # 초기 디코더 입력을 생성합니다(각 문장을 SOS 토큰으로 시작합니다)
    decoder_input = torch.LongTensor([[SOS_token for _ in range(batch_size)]])
    decoder_input = decoder_input.to(device)

    # 디코더의 초기 은닉 상태를 인코더의 마지막 은닉 상태로 둡니다
    decoder_hidden = encoder_hidden[:decoder.n_layers]
```

2

```
# 이번 반복에서 teacher forcing을 사용할지를 결정합니다
use_teacher_forcing = True if random.random() < teacher_forcing_ratio else False

# 배치 시퀀스를 한 번에 하나씩 디코더로 포워드 패스합니다
if use_teacher_forcing:
    for t in range(max_target_len):
        decoder_output, decoder_hidden = decoder(
            decoder_input, decoder_hidden, encoder_outputs
        )
        # Teacher forcing 사용: 다음 입력을 현재 목표의 출력으로 둡니다
        decoder_input = target_variable[t].view(1, -1)
        # 손실을 계산하고 누적합니다
        mask_loss, nTotal = maskNLLLoss(decoder_output, target_variable[t], mask[t])
        loss += mask_loss
        print_losses.append(mask_loss.item() * nTotal)
        n_totals += nTotal
else:
    for t in range(max_target_len):
        decoder_output, decoder_hidden = decoder(
            decoder_input, decoder_hidden, encoder_outputs
        )
        # Teacher forcing 미사용: 다음 입력을 디코더의 출력으로 둡니다
        _, top1 = decoder_output.topk(1)
        decoder_input = torch.LongTensor([[top1[i][0] for i in range(batch_size)]])
        decoder_input = decoder_input.to(device)
        # 손실을 계산하고 누적합니다
        mask_loss, nTotal = maskNLLLoss(decoder_output, target_variable[t], mask[t])
        loss += mask_loss
        print_losses.append(mask_loss.item() * nTotal)
        n_totals += nTotal

# 역전파를 수행합니다
loss.backward()

# 그라디언트 클리핑: 그라디언트를 제자리에서 수정합니다
_ = nn.utils.clip_grad_norm_(encoder.parameters(), clip)
_ = nn.utils.clip_grad_norm_(decoder.parameters(), clip)

# 모델의 가중치를 수정합니다
encoder_optimizer.step()
decoder_optimizer.step()

return sum(print_losses) / n_totals
```

• 생성 기반 챗봇 구성

1

```
def trainIters(
    model_name, voc, pairs,
    encoder, decoder, encoder_optimizer, decoder_optimizer,
    embedding, encoder_num_layers, decoder_num_layers,
    save_dir, n_iteration, batch_size, print_every, save_every, clip, corpus_name, loadFilename):

    # 각 단계에 대한 배치를 읽어옵니다
    training_batches = [batch2TrainData(voc, [random.choice(pairs) for _ in range(batch_size)])
                        for _ in range(n_iteration)]

    # 초기화
    print('Initializing ...')
    start_iteration = 1
    print_loss = 0
    if loadFilename:
        start_iteration = checkpoint['iteration'] + 1
```

2

```
# 학습 루프
print("Training...")
for iteration in range(start_iteration, n_iteration + 1):
    training_batch = training_batches[iteration - 1]
    # 배치에서 각 필드를 읽어옵니다
    input_variable, lengths, target_variable, mask, max_target_len = training_batch

    # 배치에 대해 학습을 한 단계 진행합니다
    loss = train(input_variable, lengths, target_variable, mask, max_target_len, encoder,
                 decoder, embedding, encoder_optimizer, decoder_optimizer, batch_size, clip)
    print_loss += loss

    # 경과를 출력합니다
    if iteration % print_every == 0:
        print_loss_avg = print_loss / print_every
        print("Iteration: {}; Percent complete: {:.1f}%; Average loss: {:.4f}".format(
            iteration, iteration / n_iteration * 100, print_loss_avg))
        print_loss = 0

    # Checkpoint를 저장합니다
    if (iteration % save_every == 0):
        directory = os.path.join(save_dir, model_name, corpus_name, '{}-{}-{}'.format(
            encoder_num_layers, decoder_num_layers, hidden_size))
        if not os.path.exists(directory):
            os.makedirs(directory)
        torch.save({
            'iteration': iteration,
            'en': encoder.state_dict(),
            'de': decoder.state_dict(),
            'en_opt': encoder_optimizer.state_dict(),
            'de_opt': decoder_optimizer.state_dict(),
            'loss': loss,
            'voc_dict': voc.__dict__,
            'embedding': embedding.state_dict()
        }, os.path.join(directory, '{}_{}.tar'.format(iteration, 'checkpoint')))
```

• 생성 기반 챗봇 결과

Python - split() 결과

Dropout 레이어를 평가 모드로 설정합니다

```
encoder.eval()  
decoder.eval()
```

탐색 모듈을 초기화합니다

```
searcher = GreedySearchDecoder(encoder, decoder)
```

채팅을 시작합니다 (다음 줄의 주석을 제거하면 시작해볼 수 있습니다)

```
evaluateInput(encoder, decoder, searcher, voc)
```

> 안녕하세요?

Error

> 반갑습니다.

Bot: 저도 반가워요. 좋은 사람 만나세요. 생각하고 말하는 건 어떨까요. 좋은 사람

> 이름이 뭐예요?

Bot: 지금도 그래요. 좋겠네요. 해보세요. 말해보세요. 수 있을 거예요. 더

> 어디에 있나요?

Error

> 보고싶네요.

Error

> 보고싶습니다.

Error

> 집에 가고 싶어요.

Bot: 집이 하고 싶은 걸 찾아보세요. 가요. 할 수 있어요. 더 사랑해주세요.

> 사랑하는 사람한테 가고 싶어요.

Bot: 언젠가 예쁜 사랑이 더 커지거나 깊어지지 않아요. 깊게 만들어 주기도 할 거예요. 할

> 사랑해요.

Bot: 어떤 말도 위로가 되지 않겠지만 힘내세요. 있어요 좋은 것 같아요. 주지 마세요.

KoNLPy – komoran() 결과

Dropout 레이어를 평가 모드로 설정합니다

```
encoder.eval()  
decoder.eval()
```

탐색 모듈을 초기화합니다

```
searcher = GreedySearchDecoder(encoder, decoder)
```

채팅을 시작합니다 (다음 줄의 주석을 제거하면 시작해볼 수 있습니다)

```
evaluateInput(encoder, decoder, searcher, voc)
```

> 안녕하세요?

Bot: 살짝 는 못 쓰 게 되 ㄴ다면 솔직 하 게 고백 하 아 보 시 어요 . 죠 ?

> 안녕하세요.

Bot: 안녕하세요 . 맘 을 먹 는 건 어떨 ㄹ까요 ? 하 어요 . 지 않 을까요 ?

> 안녕하세요.

Bot: 안녕하세요 . 맘 을 먹 는 건 어떨 ㄹ까요 ? 하 어요 . 지 않 을까요 ?

> 만나서 반가워요.

Bot: 너무 좋아하 았 던 게 아니 라면 잘못 의 사랑 을 좋아하 나 보 아요 . 었 나 요 .

> 살고싶지않아요.

Bot: 당신 은 당신 의 마음 을 읽 을 수 있 는 거 예요 . 지 않 을까요 ?

> 보고싶어요.

Bot: 보 고 싶 은 건 없 나요 . 거 예요 . 지 ㄴ 않 을 거 예요 .

> 보 고 싶 어 요.

Bot: 친구 들 과 연락 하 아 보 시 어요 . 었 나요 ? . 되 ㅸ니다

> quit

- **생성 기반 챗봇 결과**

아내가 이제 건강해져서 여행도 다니고 있어. 그런 모습을 보는 것이 참 기뻐.
아내가 건강해져서 여행도 다니실 수 있게 되다니 참 다행이에요.

그냥 공무원 친구들의 노력을 생각하면서 인정해주려고. 개들이 노력한 만큼의 대
가를 받는 거겠지 뭐. 당신의 생각대로 잘 돼서 질투심을 덜길 바라요.

생각도 못했는데 현명한 아내를 뒀서 너무 고마워. 아내에게 고마운 마음을 어떻게
전하면 좋을까요?

오래 사귄 여자 친구와는 처음 만날 때처럼 떨림은 없지만 편안한 안정감을 느낄
수 있어. 오래 사귄 여자친구에게 편안한 안정감을 느껴서 행복하시군요!

응. 너무 편하다. 다음 달 대출 상황도 연체되지 않게 열심히 일해야겠어. 열심
히 일해서 다음 달 대출 상황도 기한을 지키길 바라요.

다 내 잘못이지 아니에요. 너무 자책하지 마세요.

착해서 잘해주는 건지 좋아하는 건지 헛갈린다고 말해보세요.

선생님께 따돌림 사실을 진작 알렸다면 이렇게까지 심각해지지 않았을지도 모를 것
같아. 요새 너무 슬퍼. 일찍 선생님께 말씀 드리지 못한 것이 후회되시나 봐요.

나이를 먹으면서 몸 관리를 소홀하게 했더니 이렇게 빨리 늙어버렸네. 다른
일로 바쁘셨나 봐요. 지금이라도 운동하며 관리 하면 어떨까요?

주말은 근무시간이 아닌데 자꾸 회사 사람들이 불러내서 짜증나. 주말에 회사
사람들 때문에 쉬질 못하여서 화가 나셨군요. 조금 더 자세히 말씀해주시겠어요?

- AI-Hub에서 다운로드 받은 데이터는 너무 구어체스럽기 때문에 챗봇이 이상한 대답을 출력한다.

- 결과 비교

시나리오형 챗봇 결과

```
chat(intent_dict)
```

```
주문하시겠습니까? > 안녕하세요, 음...빅맥세트하나에 음료수는 배 칠러로 주세요.  
요청하신 ['빅맥'], ['배 칠러'], []-['세트'] [] [] 주문이 완료되었습니다.  
주문을 이어서 하시겠습니까? > 네  
주문하시겠습니까? > 1955버거 단품도 추가해주세요  
요청하신 ['일구오오'], [], []-['단품'] [] [] 주문이 완료되었습니다.  
주문을 이어서 하시겠습니까? > 네  
주문하시겠습니까? > 오레오 맥플러리도 하나 부탁드립니다.  
요청하신 [] [] ['오레오 맥플러리'] 주문이 완료되었습니다.  
주문을 이어서 하시겠습니까? > 아니요  
주문이 완료되었습니다.
```

- 정해진 스크립트에 맞춰서 출력하기 때문에 사용자 사전만 잘 구축되어 있다면 문제없이 출력한다.

생성 기반 챗봇 Python - split() 결과

```
> 안녕하세요?  
Error  
> 반갑습니다.  
Bot: 저도 반가워요. 좋은 사람 만나세요. 생각하고 말하는 건 어떨까요. 좋은 사람
```

- Space 단위로 쪼개어 학습을 시켰기 때문에 종종 Error를 출력한다.

생성 기반 챗봇 KoNLPy - komoran() 결과

```
> 안녕하세요?  
Bot: 살짝 는 못 쓰 게 되 ㄴ다면 솔직 하 게 고백 하 아 보 시 어요 . 죠 ?  
> 안녕하세요.  
Bot: 안녕하세요 . 맘 을 먹 는 건 어떨 ㄹ까요 ? 하 어요 . 지 알 을까요 ?  
...
```

- 형태소 단위로 쪼개어 학습했기 때문에 출력은 이상할지라도 문맥은 Space 단위로 학습했을 때보다 우수하다.

- 결론

지금까지 진행한 프로젝트의 주된 내용을 정리하자면 다음과 같다.

- 1. 시나리오형 챗봇 without Deep Learning
- 2. 생성 기반 챗봇 with Deep Learning

커스터마이징 된 시나리오형 챗봇 같은 모델은 데이터 전처리와 사용자-챗봇 간 대화 알고리즘 그리고 사전 부분이 주된 알고리즘이다. 특히 사전 부분과 데이터 전처리는 거의 동시에 진행이 되어야 하므로 현재 구현된 챗봇 모델보다 개선되어야 할 부분이 많은 것은 사실이다. 하지만 특정 유저군에게만 사용한다는 전제하에 빠르게 구현할 수 있고 사전만 잘 구축된다면 딥러닝보다 우수한 효율을 보일 수 있을 것이다.

하지만 사용자의 인텐트와는 별개의 문자가 오고 갈 경우에는 별도의 알고리즘이 필요하며, 커스터마이징을 벗어난 자유대화 형식 챗봇의 경우는 시나리오형 챗봇으로는 한계를 보일 수밖에 없다. 따라서 사용자의 실생활에 밀접하게 접근하고자 하는 챗봇 서비스를 도입할 경우에는 생성 기반 챗봇(자유대화 형식)에 딥러닝을 적용함과 동시에 사용자 개개인에 맞게 커스터마이징을 해야 한다. 이는 현 프로젝트의 범위를 벗어나며 난도 또한 급격하게 높아지므로 구현하지 못 했다.

Pytorch Tutorial을 참고하여 Attention Mechanism을 챗봇에 적용하여 보다 깊은 네트워크에서 Greedy Search 알고리즘과 함께 우수한 성능을 보이는 생성 기반 챗봇을 구현했다. 앞서 보인 결과처럼 챗봇이 엉뚱한 답변을 보였지만, 이는 데이터를 제대로 전처리하지 않았고 데이터의 성질(기쁨, 슬픔 등)을 전혀 고려하지 않은 상태에서 학습을 진행했기 때문이다. 하지만 구어체가 대부분이었음을 생각한다면(약 20,000 문장 이상) 문맥은 이상하더라도 답변의 첫 문장은 제대로 답변하는 모습을 보였고 두 번째 문장부터는 구어체의 모습을 학습한 결과를 보였다.