Gluon으로구현해보는 한영기계번역 모형

저자 : 전희원, 편집자 : 조병승(마이크로소프트웨어)

새로운 딥러닝 프레임웍으로 각광을 받고 있는 Gluon을 기반으로 간단하게 한영기계번역 모형을 구축해 보도록 하겠다. 기초부터 설명하기엔 너무 방대해 RNN, Fully Connected 레이어 등 딥러닝의 기본 개념에 대해서 익숙한 독자라는 가정을 하고 설명을 진행할 예정이며, 이러한 조건에 충족하지 못한 분들은 시중에 나와있는 딥러닝 서적을 통해 학습을 한 뒤 살펴보기를 바란다.

왜 Gluon인가?

필자가 TensorFlow를 작년 년초부터 실무에 적용하면서 시행착오를 상당히 많이 해왔다. 물론 사용해본 경험이 있는 독자분들은 잘 아시겠지만, API가 직관적이지 않고 딥러닝 영역도 생소하고 또한 무엇보다 디버깅이 어려워 정확하게 내가 원하는 방향으로 모형을 만들었다는 확신이 들기까지는 상당한 시간이 소요되었다. 그러한 와중에 Keras를 만나게 되었는데, 위와 같은 단점이 한순간에 사라져 버리는 마법과 같은 경험을 하였다. 게다가 상당히 성숙한 프레임웍이면서 동시에 상당히 많은 자료가 인터넷에 누적되어 있어 실무에서는 운영 활용도가 높은 Keras를 기반으로 많은 모형을 작성하고, 활용하였다.
Keras를 만난건 딥러닝의 파워를 좀더 쉽게 접근하게 하는 큰 역할을 수행하였고, 이후 딥러닝 공부를 하는데 큰 동기 부여를 해준 고마운 프레임웍이라 이야기 할 수 있겠다.

이러한 과정중에 개인적인 딥러닝 기반 NLP 공부를 시작하게 되었는데, NLP 모델링 방식 중에 하나인 seq2seq(뒤에 자세한 설명 예정)을 구현하는데, 큰 어려움을 겪게 되면서 일종의 대안 프레임웍을 고민하게 되었다. seq2seq 모형이 기존의 딥러닝 모형과 다른점은 학습의 네트웍 구조와 예측시의 네트웍 구조가 다르다는 점이었고, 이를 정확하게 Keras로 구현하는게 상당한 난이도가 있는 작업이었다.

결국 Keras에서 이젠 졸업을 해야 될 때가 된것으로 판단하고 모형 구현과 디버깅이 직관적인 프레임웍을 찾던중 Gluon을 발견하게 되었다.

from : freesearch.pe.kr

사실 이 프레임웍은 버전이 0.1밖에 되지 않은것으로 처음 사용하기에는 시행착오의 부담감이 있었으나, 그 과정 자체에서 딥러닝 프레임웍에 대해서 자세히 아는 계기가 되어 오히려 실력향상에 큰 도움이 되었다. 이런 단점까지 무마하는데 결정적인 영향을 줬던 Gluon만의 장점이 존재한다.

- TensorFlow의 Symbolic 방식과 PyTorch의 Imperative 방식을 모두 지원한다.

PyTorch처럼 run by doing을 통한 직관적인 프로그래밍을 기반으로 코드를 작성(Imperative)하고 어느정도 코드가 잘 구성이 되었다는 판단이 드는 시점에 hybridize() 명령어를 통해 Symbolic 방식으로 전환해 빠르게 학습을 수행할 수 있도록 지원하고 있는데, 이는 어떠한 프레임웍도 지원하지 않는 Gluon만의 장점이다. 실제 필자의 경험으로 판단하건데 적게는 1.5배 이상 학습속도 향상을 가져왔다. 그리고 Python의 numpy와 gluon에서 쓰는 API(실제 내부적으로는 MXNet NDarray) 함수명 및 역할이 95%이상 일치하기 때문에 적응하기 용이한 장점이 있다.

Gluon에 대한 소개는 이정도로 하도록 하고 본격적으로 NMT(Neural Machine Translation)에 대한 이야기를 해보자.

기계번역의 역사는 언어학자들의 룰을 생성하고 번역모형을 작성하는 RBMT(Rule based Machine Translation)으로 시작되었으며, 어느정도 성능을 보여주기 위해서 상당한 인력과 시간이 소요되는 단점과 함께 특정 도메인에 의존한 특징때문에 확장성이 매우 떨어지는 기법이였다. 기계학습 기반의 모형이 일반화 되면서 RBMT에 대한 의존성은 떨어지게 되었는데, SMT(Statistical Machine Translation)는 대량의 코퍼스를 기반으로 번역모형과 언어모형을 생성해 이를 디코더에서 번역문장을 생성하는데 활용하는 방식으로 번역을 수행한다.
NMT는 병렬 코퍼스를 제공하면 어떠한 언어쌍이든지 심지어 다른 문법체계를 가지는 언어쌍이라도 용이하게 번역모형을 생성해주며, 모델러에게 특별한 언어학적 지식을 요구하지 않음과 동시에 탁월한 번역 성능을 보여주는 기법이다. 이러한 특징 때문에 오류수정이 어려우며, 방대한 계산을 요구하기 때문에 값비싼 GPU리소스를 활용해야만 된다.
필자는 참고로 Text Mining이나 검색쪽 경험이 있으나 기계번역은 경험이 전무하다. 그럼에도 불구하고 데이터와 GPU의 힘으로 기계번역에서 ‘hello World!’를 찍어볼 수 있었고, 그 과정에 대해서 소개하도록 하겠다.

실제 쓸만한 기계번역 모형을 구축하기 위해서는 최소 수백만 이상의 학습셋이 필요하고, 모형 아키텍처가 구비되고 난 뒤에는 거의 데이터 확보 싸움이라고 봐야 될 정도로 양질의 그리고 대량의 학습셋이 필요하다. 현재 공개된 병렬 코퍼스는 필자의 조사 결과 약 9만건 내외 정도인데, 이 정도로 상용 번역기의 성능을 내는건 불가능하다. 따라서 어느정도 번역모형의 가능성을 확인해보는 과정으로 이 글의 한계를 인지해주면 좋을것 같다.

데이터/전처리

학습셋은 이곳(https://github.com/jungyeul/korean-parallel-corpora)에서 획득한걸 사용하겠다. 디렉토리를 살펴보면 한글, 영문 코퍼스 파일이 나뉘어 있고 학습과 테스트셋도 잘 나뉘어 있는 것을 볼 수 있다. 아래 표와 같이 각 라인별로 한글 문장은 같은 라인의 영문문장과 1:1로 매핑이 된다.

	en	ko
0	Much of personal computing is about "can you t...	개인용 컴퓨터 사용의 상당 부분은 "이것보다 뛰어날 수 있느냐?"₩n
1	so a mention a few weeks ago about a rechargea...	모든 광마우스와 마찬가지 로 이 광마우스도 책상 위에 놓는 마우스 패드를 필요로 하...
2	Like all optical mice, But it also doesn't nee...	그러나 이것은 또한 책상도 필요로 하지 않는다.₩n

위 문장을 그대로 학습에 넣기는 어려워 일종의 전처리가 필요하다. 이 부분은 많은 방식이 존재하나 가장 쉽게 접근할 수 있는 방식으로 진행하고자 한다.

한글은 KoNLPy[1]의 은전한닢 형태소[2] 분석기로 토큰화(tokenization)를 시행하고 영문은 모두 소문자로 변환한 뒤 Porter stemmer[3]를 기반으로 토큰화를 진행한다. 위 문장은 아래와 같은 토큰화된 문장으로 변환된다. 그리고 한글, 영문 모두 토큰화된 문장 시작과 마지막에 문장의 시작을 의미하는 “START”와 끝을 의미하는 “END” 태그를 붙여준다.

	en	ko
0	[START, much, of, person, comput, is, about, "...	[START, 개인, 용, 컴퓨터, 사용, 의, 상당, 부분, 은, ", 이것, 보...
1	[START, so, a, mention, a, few, week, ago, abo...	[START, 모든, 광, 마우스, 와, 마찬가지, 로, 이, 광, 마우스, 도, ...
2	[START, like, all, optic, mice,, but, it, also...	[START, 그러나, 이것, 은, 또한, 책상, 도, 필요, 로, 하, 지, 않,...

이러한 방식으로 전처리를 해주는 이유는 한글 어절은 같은 의미의 어절이라더라도 많은 형태를 가지고 있기 때문이다. 만일 띄어쓰기 단위로 전처리를 했다면 같은 의미지만 수많은 형태를 가지고 있는 어절에 대해서 충분히 학습할 만한 데이터가 있어야 되는데, 현실적으로 불가능한 이야기이다. 영어의 경우도 한글만큼은 아니지만 어절의 형태가 다양한 편이다. 따라서 stemmer로 토큰화를 진행해 주는 것이다.

사전과 임베딩

어떠한 기계학습 알고리즘도 단어를 그 자체로 넣어서 예측을 하는 편의를 제공하는 알고리즘은 없다. 결국 NMT모형에 입력하기 위해서 단어 하나도 숫자로 표현되어야 되는것이다.
아래와 같이 단어(key)를 인덱스(index)를 매핑해 뉴럴넷 내부적으로 숫자로 표현될 수 있는 사전(Vocabulary)을 생성한다. 이 사전구조는 단어 인덱스의 나열로부터 문장을 생성할때 다시 사용된다.

[1] http://konlpy-ko.readthedocs.io/ko/v0.4.3/
[2] 세종 말뭉치 기반의 형태소 분석기
[3] https://pypi.python.org/pypi/stemming/1.0

from : freesearch.pe.kr

	index	key
72	0	the
24	1	END
0	2	START
55	3	.
98	4	다
37	5	이
47	6	는
92	7	을
10	8	의
69	9	to

	index	key
19192	19227	haechan
19199	19228	Koreas
19204	19229	oecd
19207	19230	지검
19208	19231	Former
19222	19232	신기남
19226	19233	meghani
19232	19234	sorry
19235	19235	__ETC__
19236	19236	__PAD__

표1. 단어와 인덱스

위 표1에서 START, END 이외의 중요한 키워드는 '__ETC__'와 '__PAD__'이다. 약 2만개의 인덱스가 존재하는데, GPU 메모리와 연산시간 그리고 학습셋이 한정되어 있기 때문에 모든 단어를 인덱스에 추가하는건 불가능하다. 따라서 특정 빈도 이하의 단어는 '__ETC__'에 할당해 학습셋에서 보지못한 키워드에 대한 고려와 더불어 컴퓨팅 한계를 피한다. 영어 1문장과 한글 1문장이 쌍으로 학습셋에 들어가게 되는데, 입력시 문장의 길이를 맞춰서 넣어야 될 필요성이 있다. 이를 위해 '__PAD__'라는 특수 단어를 지정해 정해준 길이에 미치지 못하는 문장에 대해서 남은 공간에 '__PAD__'라는 단어로 채워준다.
그럼 이 숫자를 가지고 어떻게 문장을 만들어줄 것인지를 고민해야 된다.

"아버지가 방에 들어가신다."라는 문장을 전처리 하면 아래와 같은 리스트로 변환된다.

["START", "아버지", "가", "방","에","들어가", "신다", ".","END"]

우리가 처리하고자 하는 문장의 최대 길이를 10개로 제한한다면 아래와 같은 데이터가 생성된다.

["START", "아버지", "가", "방","에","들어가", "신다", ".", "END", "__PAD__"]

위 문장을 사전과 매핑하면 아래와 같다.

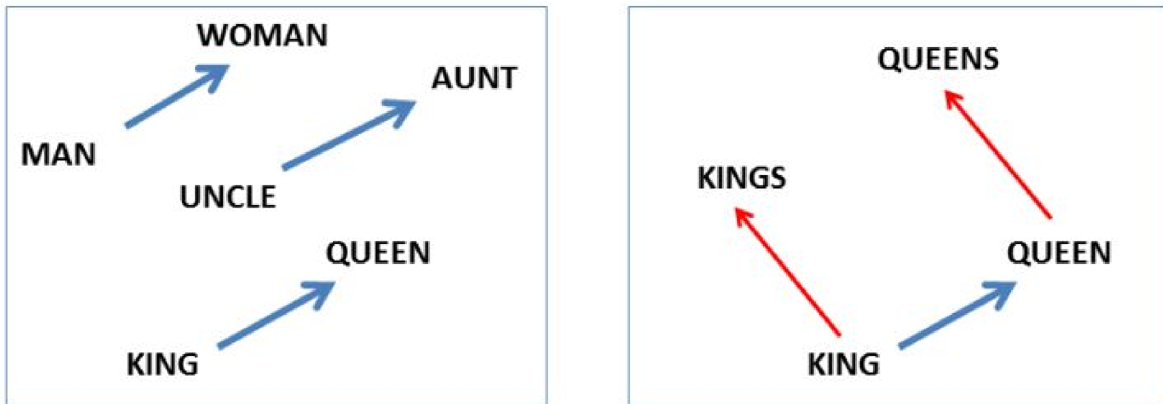
[2, 234, 34, 56, 78, 2345, 673, 3, 1, 19236, 19236]

이런 상태로 알고리즘에 입력하면 안된다. 왜냐면 알고리즘은 아버지(234)와 방(56)의 수치적 차이를 하나의 큰 정보로 활용하려 하기 때문이다. 사실 임의대로 숫자가 할당된거지 숫자의 의미는 전혀 존재하지 않는게 맞다. 이를 위해 one hot encoding이라는 기법을 사용하게 된다. 수치적 차이를 없애고 '아버지'와 '방'이라는 단어가 다르다는 것만 계산에서 고려하려 하기 위함이다.
'아버지'라는 단어를 아래와 같이 표현하는게 one hot encoding 방식이다.

0	1	2	3	4	...	...	234	235	236	...	...	19233	19234	19235	19236
0	0	0	0	0	0	0	0	0	**1**	0	0	0	0	0	0

from : freesearch.pe.kr

이 방식은 범주형 자료형에 대해서 기계학습 알고리즘에 입력할때 주로 사용되는 방법이나 차원의 개수가 매우 커지는 단점이 있고, 대부분의 값이 0인 sparse한 매트릭스를 생성하게 된다. 현재 우리가 생성한 사전의 크기가 2만 차원에 육박함으로 각 단어당 약 2만 차원의 벡터가 필요한 셈이다.
one hot encoding이 범주형 데이터에 대해서 속성화 하는데 유용한 기법이긴 하나 단어와 같이 그 차원이 클 경우엔 매우 비효율적인 방법론이 되어 버린다. 이 때문에 hashing trick[4]과 같은 방법이 활용되기도 하나 최근에 딥러닝이 활성화 되면서 나온 word2vector가 효과적인 대안으로 활용되고 있다.



(Mikolov et al., NAACL HLT, 2013)

그림1. word2vector 예제

one hot encoing이 희소(sparse)벡터로 단어를 표현했다면, word2vector는 밀집(dense)벡터로 단어를 표현함에 따라 차원이 기하급수적으로 늘어나는것을 방지할 수 있는 장점과 더불어 단어 벡터를 기반으로 유사한 단어와의 의미적 관계를 유추할 수 있다는 장점을 가지고 있다. 이때문에 단어를 기반으로 어떠한 일반화된 특징을 찾는게 가능하다. 예를 들면 위에서 King, Queen 벡터의 평균값은 일종의 왕족(Royal Family)을 의미하는 벡터라 할 수 있을 것이다. one hot encoding의 경우 단어들간에 유사도를 계산하는게 의미가 없었으나 word2vector를 활용하면 단어들간의 유사성을 기반으로 효과적인 예측모형 생성이 가능해진다. 반면 과도한 일반화로 인해 엉뚱한 예측 결과가 도출되기도 하는 부작용도 존재한다.

[4] https://en.wikipedia.org/wiki/Feature_hashing

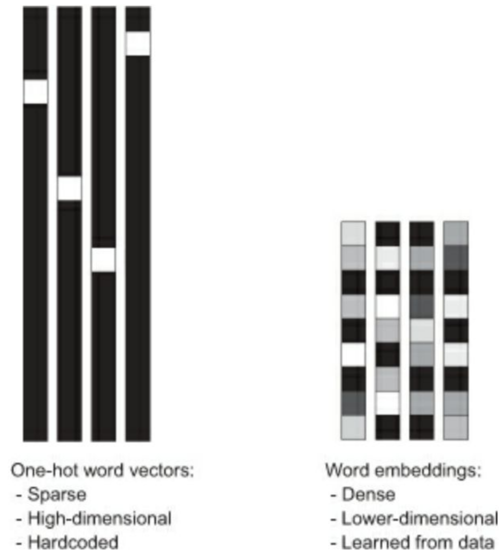


그림2 .one-hot vector와 워드 임베딩(word2vector)
from Deep Learning with Python

효과적인 word2vector를 생성하는건 전문도구를 사용하면 간단하게 생성이 가능하다. 필자의 경우 gensim[5]을 사용했다.

```
w2v.wv.similar_by_word("사과")
```

```
[('사죄', 0.6981794834136963),
 ('유감', 0.6753973960876465),
 ('거절', 0.6712061166763306),
 ('표시', 0.6687175035476685),
 ('표명', 0.6443376541137695),
 ('모욕', 0.6205172538757324),
 ('결별', 0.610889196395874),
 ('요구', 0.5891960263252258),
 ('관대', 0.5823681354522705),
 ('실수', 0.5749861001968384)]
```

"사과" 단어와 유사한 단어 리스트

일반적으로 skip-gram 방식으로 word2vector를 구축하는데, 이는 문장에서 특정 중심단어를 기준으로 주변 단어를 예측하는 모형을 학습하면서 얻어진다. 따라서 gensim과 같은 도구는 문장에서 전처리가 수행된 단어 리스트가 학습셋으로 들어간다.
필자는 적절한 word2vector를 얻기 위해 영문, 한글 문장을 쌍으로 엮어서 하나의 가중치 매트릭스를 생성했다. 예를 들어 아래의 두 한글, 영문 문장 쌍이 있을 경우 word2vector학습에는 하나의 문장으로 입력이 된다.

["START", "아버지", "가", "방","에","들어가", "신다", "." ,"END"]
+

[5] https://radimrehurek.com/gensim/

[“START”, “my”, “father”, “enters”,”the”,”room”, “.”,“END”]
=
[“START”, “아버지”, “my”,“가”,“father”, “방”, “enters”,“에” ,”the”, “들어가”,”room”, “신다” “.”,“END”]

이렇게 학습을 하는 이유는 ‘father’와 ‘아버지’는 비슷한 위치에 빈번하게 같이 발생할 것이기 때문에 벡터의 유사도가 높게 학습이 될 것이고 번역 모형의 초기 학습시 빠른 수렴을 하는데 도움이 되기 때문이다.

필자가 만들어 놓은 create_embedding() 함수를 활용하면 “embed_tr_dir” 경로에 존재하는 텍스트 파일을 기반으로 학습을 해서 ‘ko_en.mdl’, ‘ko_en.np’, ‘ko_en.dic’파일을 생성하게 된다. ‘ko_en.mdl’은 gensim에서 활용 가능한 모델 파일이고 ‘ko_en.np’는 row가 단어사전 인덱스 번호이고 컬럼이 각 단어의 벡터로 구성되는 numpy 행렬이 저장된 파일이다. ‘ko_en.dic’은 사전 인덱스와 단어사이의 매핑 정보를 포함하는 객체가 저장되어 있다. ‘ko_en.dic’을 기반으로 한글/영문문장이 인덱스의 나열로 변환이 가능해진다. min_count는 학습셋에서 고려하는 단어의 최소 출현 빈도인데, 이 빈도 이하의 단어는 고려 대상에서 빠지게 된다. iter는 학습시 전체 학습셋을 몇번 순회할 것인지를 의미하며, size는 임베딩 벡터(word2vector)의 사이즈를 의미한다. worker는 멀티코어를 활용하여 계산 효율을 높이기 위한 인자이며, 보통 머신의 코어수로 정한다. window는 한 단어의 중심에서 어느정도 거리에 있는 단어까지 주변단어로 인식해 학습할 것인지 정하는 인자이다. 이 수치는 유사도를 계산할때 어디까지를 동일한 문맥(context)로 볼 것인가와 매우 밀접한 관계가 있다. 이 기준은 하나의 문장일 수도 있고, 문단일수도 있으며 챕터일 수도 있다. 필자의 모형은 1문장 정도를 커버하는 윈도우 크기인 30으로 값을 주었다.

```
create_embeddings("embed_tr_dir", "ko_en.mdl", 'ko_en.np', 'ko_en.dic', min_count=13,
              iter=50,
              size=50, workers=10, window=30)
```

구조

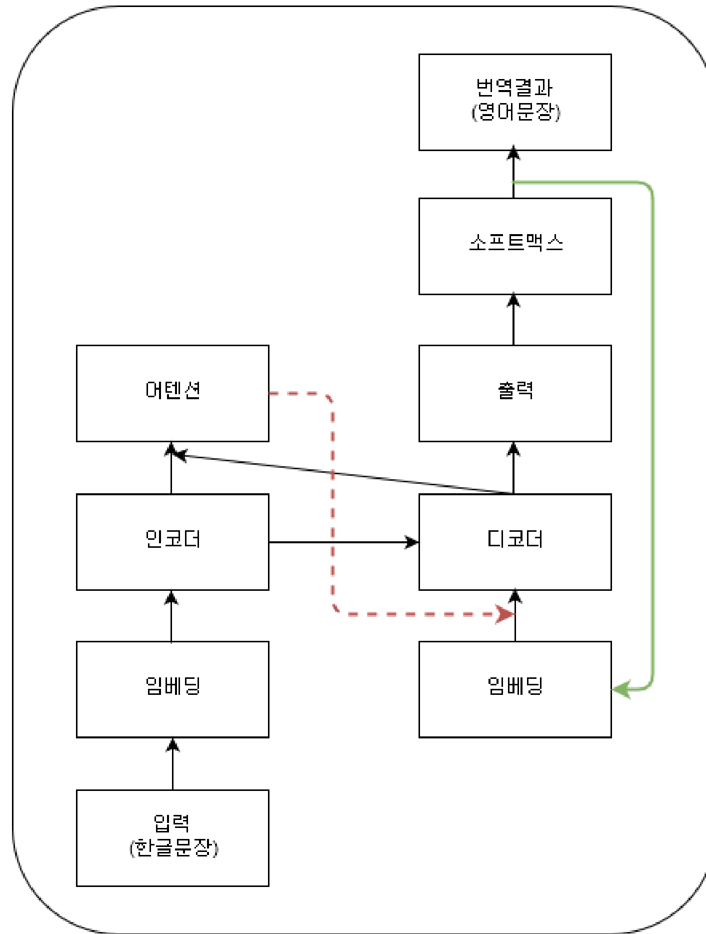


그림3. NMT 구조

입력은 아래와 같은 한글 문장을 받게 되고, 이것들이 임베딩(word2vector)의 형태로 인코더로 입력된다. 임베딩 행렬은 사전크기(약 2만) x 임베딩 차원(50)으로 구성되어 있으며 입력되는 한글 단어의 인덱스에 매핑되어 인코더에 입력되게 된다.
인코더 부분은 bidirectional GRU(Gated Recurrent Unit)로 구성이 되어 있는데, 이 부분은 그림4와 같다.

```
class korean_english_translator(gluon.HybridBlock):

    def __init__(self, n_hidden, vocab_size, embed_dim, max_seq_length,attention=False
,**kwargs):
        super(korean_english_translator,self).__init__(**kwargs)
        #입력 시퀀스 길이
        self.in_seq_len = max_seq_length
        #출력 시퀀스 길이
        self.out_seq_len = max_seq_length
        # LSTM의 hidden 개수
        self.n_hidden = n_hidden
        #사전 크기
        self.vocab_size = vocab_size
        #max_seq_length
```

from : freesearch.pe.kr

```
        self.max_seq_length = max_seq_length
        #임베딩 차원수
        self.embed_dim = embed_dim
        #어텐션 사용 여부
        self.attention = attention
        with self.name_scope():
            self.embedding = nn.Embedding(input_dim=vocab_size,
               output_dim=embed_dim, dtype="float16")
            self.encoder=
               rnn.BidirectionalCell(rnn.GRUCell(hidden_size=int(n_hidden/2)),
               rnn.GRUCell(hidden_size=int(n_hidden/2)))
            self.decoder = rnn.GRUCell(hidden_size=n_hidden)
            self.batchnorm = nn.BatchNorm(axis=2)
            #flatten을 false로 할 경우 마지막 차원에 fully connected가 적용된다.
            self.dense = nn.Dense(self.vocab_size,flatten=False)
            if self.attention:
                self.attdense = nn.Dense(self.max_seq_length,
                   flatten=False, activation='relu')
                self.attn_combine = nn.Dense( self.vocab_size,
                   flatten=False, activation='relu')
. . .
```

코드1. korean_english_translator 클래스의 생성자 파트

위 코드는 실제 모형에서 사용할 Gluon의 각종 레이어를 생성해주는 곳으로 우리가 그림3과 매핑해서 확인해보면 매우 직관적으로 정의가 되는 것을 알 수 있다.
그림3에서 임베딩에 해당되는 객체가 self.embedding이며 이는 한글/영어 모두 공용으로 활용된다. self.encoder는 인코더와 매핑되며, self.decoder는 디코더와 매핑된다.
self.attention이 True일때 생성되는 객체에는 두가지 Dense레이어가 사용되는데, 그림5에서 어텐션 박스 안에 있는 두개의 Dense 레이어와 매칭시킬 수 있을 것이다. 결국 이렇게 정의된 객체들을 결합해 어떻게 데이터를 기반으로 학습시킬지는 정의해 주는 일이 남았다.
여기서 RNN셀로 활용된 GRU셀은 기존 RNN셀의 단점으로 지적된 Vanishing Gradient문제를 해결한 셀로 LSTM에 비해서 계산복잡도가 크면서도 성능은 비슷한 특징을 가지고 있다. Vanishing Gradient는 RNN셀이 긴 문장을 처리하면서 과거의 정보를 잘 기억하지 못하는 문제를 일컷는다. 이는 시퀀스의 과거로 갈수록 gradient가 0으로 수렴해 학습을 잘 하지 못하는 이슈와 관련이 있다.
GRU는 그림4와 같이 z_t를 만들기 위해 과거정보(h_{t-1})와 현재정보(n_t)를 얼마정도 반영하여 현재정보 출력(h_t)을 만들지 결정하게 된다.

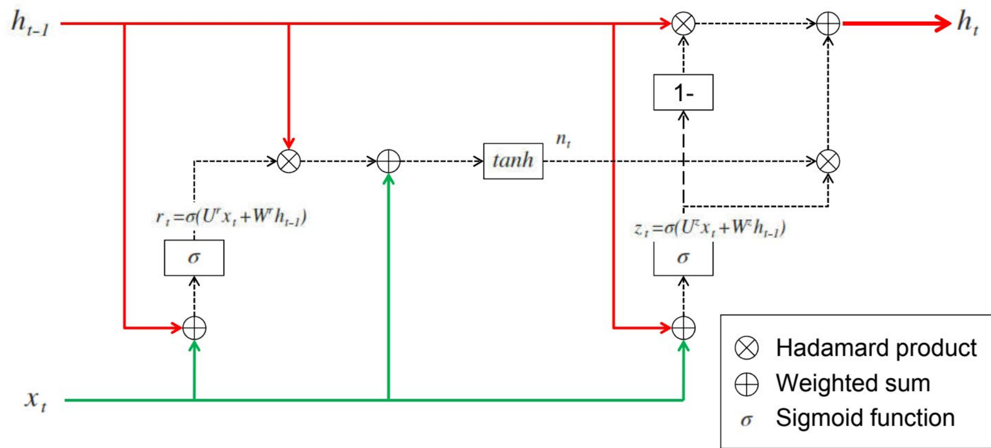


그림4. GRUCell 구조

인코더

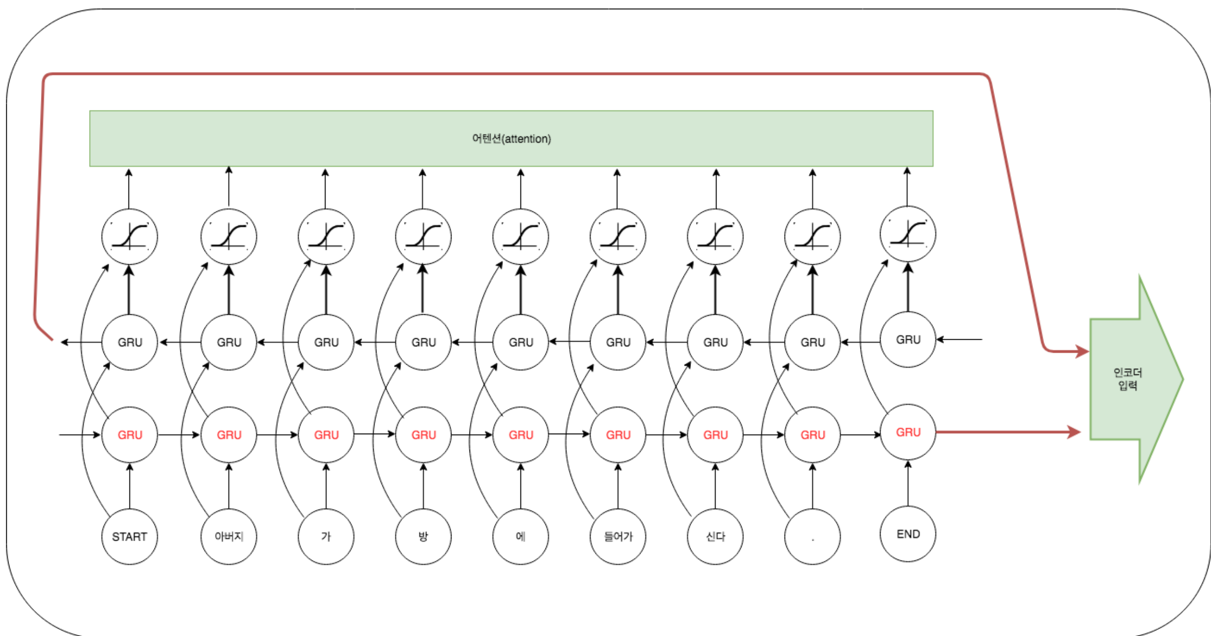


그림5. 인코더(bi-directional GRU)

self.encoder를 좀더 상세히 살펴보는 그림5와 같이 bi-directional GRU로 표현할 수 있는데, 단일 GRU를 사용하는 것 보다 문자 시퀀스를 정방향과 역방향을 모두 고려해 학습이 되기 때문에 좀더 많은 정보를 활용할 수 있게 된다.
GRU의 각 셀은 hidden state와 출력을 하나씩 리턴하는데, 시퀀스의 가장 마지막 hidden state는 인코더의 입력으로 활용되고 각 셀의 출력값은 어텐션에서 활용된다.
어텐션이 활용되는 이유는 hidden state가 문장이 길어지면 길어질 수록 먼 과거의 정보를 지니기가 어려운 구조적 이슈 때문이다.

from : freesearch.pe.kr

디코더 & 어텐션

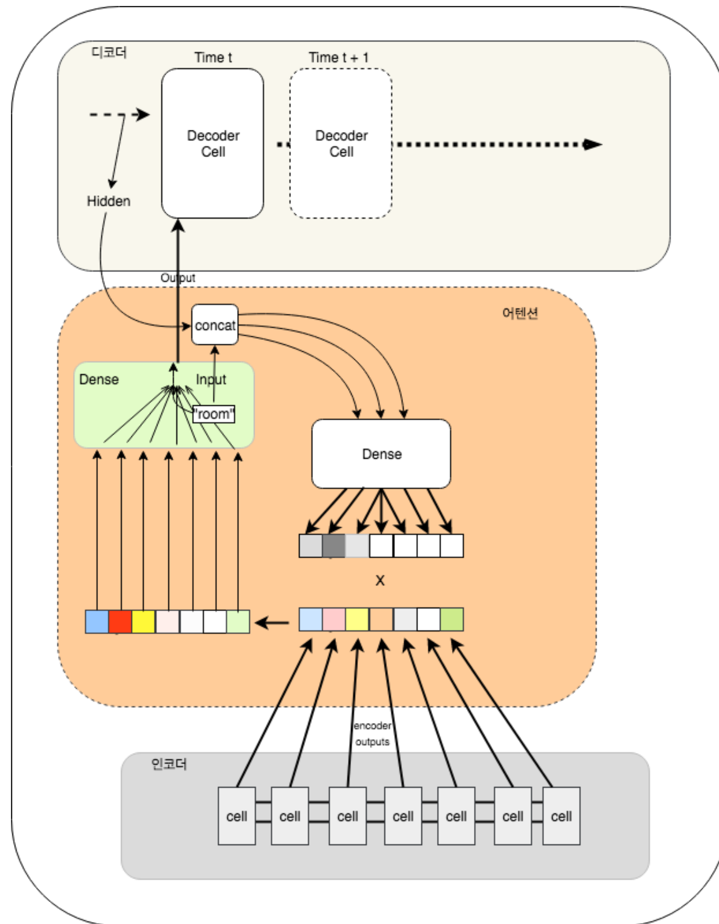


그림6. 디코어와 어텐션

앞서 인코더의 마지막 hidden state 벡터만으로 문장 스퀀스의 정보를 전달하기 어렵다는 단점을 지적했다. 또한 '방'이라는 단어와 'room'이라는 번역어의 관련성을 뉴럴넷에 지속적으로 학습을 시킬 수 있다면 한글 문장과 영어 문장의 구조적 차이를 뉴럴넷이 인지하고 번역을 하는데 활용할 수 있을 것이라는 아이디어가 바로 어텐션(attention)이다. 이를 위해 단어 'room'과 '방'이라는 단어를 연결 시키는 네트워크가 필요한데, 이 부분이 바로 하얀 박스로 되어 있는 Dense 레이어(self.attdense)와 행렬곱(F.broadcast_mul) 연산이다. 나머지 Dense레이어(self.attn_combine)는 어텐션이 가미된 디코더 입력을 생성하기 위해 일종의 차원 조정(임베딩 출력 차원과 동일하게)을 하는 역할을 수행한다.

```
def apply_attention(self, F, inputs, hidden, encoder_outputs):
        #inputs : decoder input의미
        concated = F.concat(inputs, hidden, dim=1)
        #(,max_seq_length) : max_seq_length 개별 시퀀스의 중요도
        attn_weights = F.softmax(self.attdense(concated), axis=1)
        # (N,max_seq_length,n_hidden) x (N,max_seq_length) = (N, max_seq_length,
n_hidden)
```

from : freesearch.pe.kr

```
        #attn_weigths 가중치를 인코더 출력값에 곱해줌
        w_encoder_outputs = F.broadcast_mul(encoder_outputs, attn_weights.expand_dims(2))
        #(N, vocab_size * max_seq_length), (N, max_seq_length * n_hidden) = (N, ...)
        output = F.concat(inputs.flatten(), w_encoder_outputs.flatten(), dim=1)
        #(N, vocab_size)
        output = self.attn_combine(output)
        #attention weight은 시각화를 위해 뽑아둔다.
        return(output, attn_weights)
```

결국 개별 디코더 Input에 대해서 인코더의 가중 정보를 추가해 새로운 디코더 Input을 만들어주는게 apply_attention()함수에서 수행하는 처리이다.

기계번역은 전형적인 sequence to sequence(seq2seq)문제로 문장이 입력되고 문장을 출력하는 로직으로 구성된다. 이 때문에 학습 네트워크 구조와 예측 네트워크 구조가 약간 다르다.

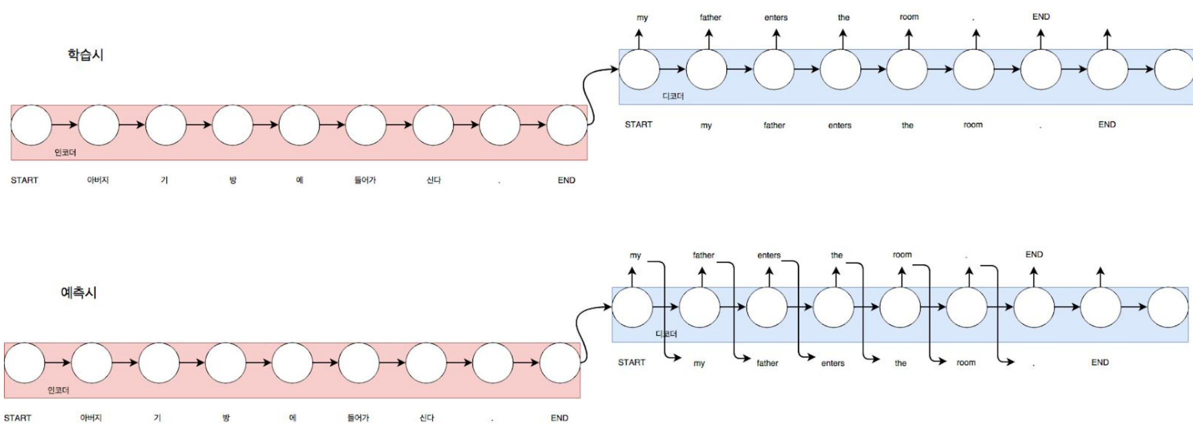


그림 7. seq2seq

간단하게 Simple RNN 1개씩 인코더 디코더에 사용해서 네트워크를 구성했다고 가정하자. 학습시에는 한글 시퀀스와 영어 시퀀스가 함께 입력이 되고 t+1 시점의 영문 시퀀스를 예측하게 디코더를 학습을 하게 된다. 이런 학습과정을 교사 강제(teacher forcing) 방법이라고 한다. 이렇게 학습된 네트워크에 대해서는 영문 시퀀스가 없이 한글 시퀀스만 주어질 것인데, 영문 시퀀스 생성시 디코더의 첫번째 셀의 입력으로 문장의 시작을 알리는 ‘START’ 태그를 입력 시키고 해당 셀이 예측한 첫번째 단어를 생성 후 이를 다시 두번째 셀의 입력으로 넣게 되는 반복을 수행한다. 출력으로 “END”나 “__PAD__”가 나올때까지 반복하고 이렇게 나온 영문 시퀀스를 결합해 문장으로 리턴하게 된다.

이런 학습에 관한 로직은 hybrid_forward()함수에 존재한다. 여기서 inputs은 한글 시퀀스가 되고, outputs은 영문 시퀀스가 된다. t+1 시점의 영문 시퀀스를 정답으로 제공되어 loss를 계산하게 된다.

```
def hybrid_forward(self, F, inputs, outputs):
        #encoder bi-directional GRU
        embeddinged_in = F.cast(self.embedding(inputs), dtype='float32')
        enout, (next_l, next_r) =
                self.encoder.unroll(inputs=embeddinged_in, length=self.in_seq_len,
```

```
                merge_outputs=True)
        next_h = F.concat(next_l, next_r, dim=1)
        embeddinged_out = F.cast(self.embedding(outputs),dtype='float32')
        #decoder GRU with attention
        for i in range(self.out_seq_len):
            #out_seq_len 길이만큼 GRUcell을 unroll하면서 출력값을 적재한다.
             p_outputs = F.slice_axis(embeddinged_out, axis=1, begin=i, end=i+1)
             p_outputs = F.reshape(p_outputs, (-1, self.embed_dim))
             # p_outputs = outputs[:,i,:]
             # 위와 같이 진행한 이유는 hybridize를 위함
             if self.attention:
                 p_outputs, _ = self.apply_attention(F=F,
                   inputs=p_outputs, hidden=next_h, encoder_outputs=enout)
                deout, (next_h,) = self.decoder(p_outputs, [next_h,] )
                if i == 0:
                    deouts = deout
                else:
                    deouts = F.concat(deouts, deout, dim=1)
        #2dim -> 3dim 으로 reshape
        deouts_orig = F.reshape(deouts,
          (-1, self.out_seq_len, self.n_hidden))
        deouts = self.batchnorm(deouts_orig)
        deouts_fc = self.dense(deouts)
        return(deouts_fc)
```

예측에 대한 코드는 아래 calculation()함수에 구현했다. hybrid_forward()함수보다 다소 복잡해 보이는 이유는 입력(input_str) 출력이 일반 문자열이어서 문자 전/후처리 이슈가 있는 것과 Y_init 변수를 통해 초기 'START'태그를 할당해 출력을 받고 해당 출력을 다시 다음 디코더 셀에 입력으로 넣는 반복코드가 있기 때문이다. 아래 코드를 그림6의 예측쪽 도표와 매칭해서 이해하면 좀더 빠른 이해가 될 것이다.
이 이외에 배치정규화(self.batchnorm)를 하기 전에 차원을 늘이고 줄이는건 hybrid_forward()함수때에는 시퀀스 길이라는게 차원에 존재했으나, 디코더셀에서 특정 t의 아웃풋을 가지고 끝까지 진행해 예측값을 도출해야 되기 때문에 시퀀스 길이가 누락되어 늘려주고 다시 원상태로 돌려주는 추가 코드가 필요했다.

```
def calculation(self, input_str, ko_dict, en_dict, en_rev_dict, ctx=mx.gpu(0)):
        #앞뒤에 START,END 코드 추가
        input_str = [['START', ] + mecab.morphs(input_str.strip()) + ['END', ],]
        X = encoding_and_padding(input_str, ko_dict, max_seq=self.max_seq_length)
        #string to embed
        X = F.cast(self.embedding(F.array(X, ctx=ctx)), dtype='float32')

        #인코더 출력값을 도출한다.
        enout, (next_l, next_r) = self.encoder.unroll(inputs=X, length=self.in_seq_len,
                                                      merge_outputs=True)
        next_h = F.concat(next_l, next_r, dim=1)
        #디코더의 초기 입력값으로 넣을 'START'를 임베딩한다.
        Y_init = F.array([[en_dict['START'],],], ctx=ctx)
        Y_init = F.cast(self.embedding(Y_init),dtype='float32')
```

```
        deout = Y_init[:,0,:]

        #출력 시퀀스 길이만큼 순회
        for i in range(self.out_seq_len):
            if self.attention:
                #print(deout.shape)
                deout, att_weight = self.apply_attention(F=F,
                  inputs=deout, hidden=next_h, encoder_outputs=enout)
                if i == 0:
                    att_weights = att_weight
                else:
                    att_weights = F.concat(att_weights,att_weight,dim=0)
            deout, (next_h, ) = self.decoder(deout, [next_h, ])
            #batchnorm을 적용하기 위해 차원 증가/원복
            deout = F.expand_dims(deout,axis=1)
            deout = self.batchnorm(deout)
            #reduce dim
            deout = deout[:,0,:]
            #'START'의 다음 시퀀스 출력값도출
            deout_sm = self.dense(deout)
            #print(deout_sm.shape)
            deout = F.one_hot(F.argmax(F.softmax(deout_sm, axis=1), axis=1),
                  depth=self.vocab_size)
            #print(deout.shape)
            #decoder에 들어갈 수 있는 형태로 변환(임베딩 적용 및 차원 맞춤)
            deout = F.argmax(deout, axis=1)
            deout = F.expand_dims(deout, axis=0)
            deout = F.cast(self.embedding(deout)[:,0,:],dtype='float32')
            gen_char  = en_rev_dict[F.argmax(deout_sm,
                  axis=1).asnumpy()[0].astype('int')]
            if gen_char == '__PAD__' or gen_char == 'END':
                    break
            else:
            if i == 0:
                    ret_seq = [gen_char, ]
            else:
                    ret_seq += [gen_char, ]
        return(" ".join(ret_seq), att_weights)
```

인코더에서 계속 문자을 생성하다가 '__PAD__' 나 'END' 태그를 만나면 생성을 종료하고 생성된 시퀀스를 반환하게 된다.

학습

학습을 하기 위해 정해줘야 되는 여러가지 파라메터가 존재한다. 첫번째로 데이터를 한번 학습에 몇문장을 넣을지 결정해야 되는데, 이를 배치사이즈라고 한다. Gluon은 DataLoader API를 통해서 입력을 받게 되기 때문에 우리가 가지고 있는 전처리된 numpy 행렬을 아래와 같이 처리해서 준비한다.

```
tr_set = gluon.data.ArrayDataset(ko_train_x, en_train_x, en_train_y)
tr_data_iterator = gluon.data.DataLoader(tr_set, batch_size=100, shuffle=True)
```

from : freesearch.pe.kr

한번에 100문장을 넣기 위해서는 GPU 디바이스 당 최소 8GB의 메모리를 가지는 두장 이상의 GPU카드가 필요하다. 만일 GPU가 1장이라면 60이하로 넣어야 학습을 할 수 있을 것이다. 필자의 경우 GTX 1080 두장을 가지고 학습을 했기 때문에 100으로 잡았다.

```
GPU_COUNT = 2
ctx= [mx.gpu(i) for i in range(GPU_COUNT)]

def model_init():
        #모형 인스턴스 생성 및 트래이너, loss 정의
        #n_hidden, vocab_size, embed_dim, max_seq_length
        model = korean_english_translator(n_hidden, vocab_size, embed_dim,
            max_seq_length,attention=True)
        model.collect_params().initialize(mx.init.Xavier(), ctx=ctx)
        #초기화 이후 임베딩은 기 학습된 임베딩 행렬을 적용한다.
        model.embedding.weight.set_data(embed_weights)
        #Rmsprop 옵티마이저 사용
        trainer = gluon.Trainer(model.collect_params(), 'rmsprop')
        #(batch size, max_seq_length, vacab_size) 매트릭스에서
        #가장 마지막 차원에 softmax crossentropy loss를 적용한다.
        loss = gluon.loss.SoftmaxCrossEntropyLoss(axis = 2)
        return(model, loss, trainer)
```

여기서 다른 초기화 코드와 다른 부분은 gensim으로 학습된 numpy 행렬값을 self.embedding 객체에 가중치로 할당하는 것이다. 최적화된 초기 가중치값이므로 이렇게 하지 않는 것보다 빠른 수렴속도를 보여준다. 나머지 부분은 주석을 참고하도록 한다.

학습 지표는 학습셋과 테스트셋의 loss로 아래와 같이 설정한다.

```
def calculate_loss(model, data_iter, loss_obj, ctx=ctx):
    test_loss = []
    for i, (x_data, y_data, z_data) in enumerate(data_iter):
        x_data_l = gluon.utils.split_and_load(x_data, ctx,  even_split=False)
        y_data_l = gluon.utils.split_and_load(y_data, ctx,  even_split=False)
        z_data_l = gluon.utils.split_and_load(z_data, ctx,  even_split=False)
        with autograd.predict_mode():
            losses = [loss_obj(model(x, y), z)
                  for x, y, z in zip(x_data_l, y_data_l, z_data_l)]
        curr_loss = (mx.nd.mean( losses[0] ).asscalar() +
            mx.nd.mean( losses[1] ).asscalar())/2
        test_loss.append(curr_loss)
    return(np.mean(test_loss))
```

GPU 두장으로 학습을 할 경우 동일한 모델 파라메터가 각 GPU카드에 설정된다. 그러니까 동일한 번역모형 한개씩 개별 GPU카드에 올라가게 된다. 100개의 문장이 두 카드에 나뉘어 들어가서 loss가 계산되고 이들 loss를 기반으로 연전파 알고리즘을 통해 gradient를 계산해 두 GPU의 모형을 모두 업데이트 하게 된다. 일종의 데이터 병렬을 통해 학습 속도 향상을 하게 되는 것이다. 여기서 두 GPU에 거의 동일한 비율로 학습셋이 들어갈 필요가 있는데, 이를 자동으로 해주는 함수가 바로 gluon.utils.split_and_load() 함수이다. 함수명대로 학습셋을 쪼개줄 뿐만 아니라 각 GPU context에 로딩까지 자동으로 해준다. 그렇게 입력된

데이터로 개별 GPU는 예측을 하게 되고 그 결과로 loss를 계산해 개별 GPU loss를 평균하여 입력 데이터의 loss를 계산해 반환한다.

```
def train(epochs, tr_data_iterator, model, loss, trainer,
          ctx= [mx.gpu(i) for i in range(GPU_COUNT)],
          mdl_desc="k2e_model", decay=False):
    ### 학습 코드
    tot_test_loss = []
    tot_train_loss = []
    for e in range(epochs):
        tic = time.time()
        # Decay learning rate.
        if e > 1 and decay:
            trainer.set_learning_rate(trainer.learning_rate * 0.5)
        train_loss = []
        for i, (x_data, y_data, z_data) in enumerate(tqdm(tr_data_iterator)):
            x_data_l = gluon.utils.split_and_load(x_data, ctx, even_split=False)
            y_data_l = gluon.utils.split_and_load(y_data, ctx, even_split=False)
            z_data_l = gluon.utils.split_and_load(z_data, ctx, even_split=False)

            with autograd.record():
                losses = [loss(model(x, y), z)
                    for x, y, z in zip(x_data_l, y_data_l, z_data_l)]
                for l in losses:
                    l.backward()
                trainer.step(x_data.shape[0])
                curr_loss = (mx.nd.mean( losses[0] ).asscalar() +
                    mx.nd.mean( losses[1] ).asscalar())/2
                train_loss.append(curr_loss)
                mx.nd.waitall()
        #caculate test loss
        test_loss = calculate_loss(model, te_data_iterator, loss_obj = loss, ctx=ctx)
        print('[Epoch %d] time cost: %f'%(e, time.time()-tic))
        print("Epoch %s. Train Loss: %s, Test Loss : %s" %
            (e, np.mean(train_loss), test_loss))
        tot_test_loss.append(test_loss)
        tot_train_loss.append(np.mean(train_loss))
        model.save_params("{}_{}.params".format(mdl_desc, e))
        return(tot_test_loss, tot_train_loss)
```

위 코드는 학습을 수행하는 코드이다.
특징적인 부분은 autograd.record() 부분으로 실제 이 영역에서 모형의 가중치 값이 업데이트 된다. 학습이 진행되면서 학습셋의 loss 뿐만 아니라 테스트셋의 loss까지 계산해 출력하주고, 마지막 각 에폭(epoch)마다 모형을 디스크에 저장하는 코드를 수행한다. 중간 중간 loss가 어떻게 도출되었는지 궁금하다면 코드 중간에 print()문을 넣어 출력해보면 된다. 보고 싶은 어떠한 수치라도 학습중에 뽑아볼 수 있다는건 딥러닝 학습에 큰 도움을 준다. 몇몇 딥러닝 프레임웍은 이런 부분이 상당히 불편해 디버깅도 어렵고, 내부 로직을 들여다 보기 어려워 경험치를 높이는데 다소 어려운 부분이 있는데, Gluon은 그런 불편함이 전혀 없다.

from : freesearch.pe.kr

필자의 경우 Rmsprop[6] 옵티마이저로 10에폭을 학습하고 이후 5에폭을 SGD(learning rate : 0.01, wd:1e-5, learning rate decay :0.5)로 수행하였다. 학습에만 약 2시간이 소요되었다.

학습 결과 테스트

몇몇 문장을 입력하고 번역된 결과를 출력해 보았다. 다소 9만건의 학습셋이 정치/경제 관련 기사라 아무래도 해당 문맥에 맞는 문장을 넣어 보았지만 학습셋에는 존재하지 않는 문장들이다.

```
eng_seq,att_weight  = model.calulation("그는 좋은 사람이다.", ko_dict=w2idx, en_dict=w2idx, en_rev_dict=idx_2w)
eng_seq

'he is a great __ETC__'
```

```
eng_seq,att_weight  = model.calulation("오바마는 미국의 대통령이다.", ko_dict=w2idx, en_dict=w2idx, en_rev_dict=idx_2w)
eng_seq

'obama is current in a new president.'
```

```
eng_seq,att_weight  = model.calulation("북한은 핵무기로 전쟁을 준비한다.", ko_dict=w2idx, en_dict=w2idx, en_rev_dict=idx_2w)
eng_seq

'north korea test a nuclear weapon program.'
```

'__ETC__'로 표현되는 것으로 볼때 '좋다'라는 단어에 적합한 단어를 번역기가 찾기 못한 것으로 판단된다. 하지만 학습셋에 해당 단어가 있는 것으로 볼때 '좋다'라는 단어가 쓰이는 다양한 용례를 모형이 찾기 못한게 아닐까 하는 생각을 해본다. 이러한 번역기의 단점을 수정하는 작업은 일반 룰 기반의 번역기보다 매우 어려운 작업이기 때문에 일반적으로 수많은 데이터를 학습하는 방향으로 해결하려는 경향이 크다.

마무리

모든 코드를 지면에 올리긴 어려워 예시로 사용된 코드를 이곳[7]에 올려두었다.
실제 번역 결과를 보면 시중에 나와있는 구글 번역기하고는 비교할 수 없는 정도라는 생각이 들 것이다. 물론 이 모형은 공개된 9만건의 학습셋으로 가능성 여부정도를 타진하기 위해서 구축된 것이여서 비교의 대상은 아니다. 상용의 경우 적게는 수백만건 많게는 수천,억 만건의 학습데이터가 사용되며 여기서 구현된 것 말고도 많은 튜닝이 되어 있는 모형이기 때문이다. 그렇지만 이정도의 데이터만으로 문맥을 어느정도 타깃하고 문장을 만들어 낸다는 느낌을 받기에는 충분했을 것이다. 아마도 이런 마법과 같은 경험을 하는 재미가 딥러닝을 하는 가장 큰 재미지 않을까 하는 생각을 해본다.

---

[6] https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf
[7] https://github.com/haven-jeon/ko_en_neural_machine_translation