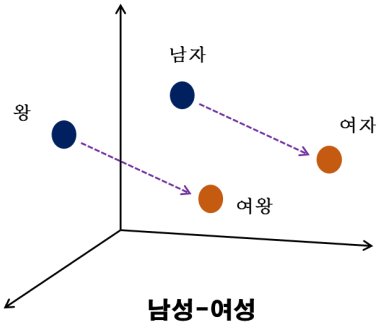
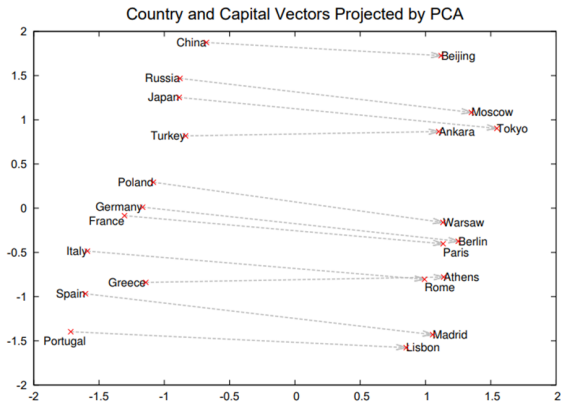
11. 단어 임베딩

전통적으로 자연어 처리에서는 단어를 의미나 발음을 무시하고 각각을 개별적인 기호로 취급한다. 단어를 벡터로 나타낼 때는 총 단어 수만큼의 길이의 벡터에서 다른 모든 값은 0으로 하고 단어 번호에 해당하는 원소만 1로 표시한다. 예를 들어 '학교', '컴퓨터', '집' 3단어만 있고 순서대로 1번, 2번, 3번이라면 학교는 $(1, 0, 0)$, 컴퓨터는 $(0, 1, 0)$, 집은 $(0, 0, 1)$로 나타내는 것이다. 이를 one-hot encoding이라고 한다. 단어 문서 행렬의 각 행, 즉 문서를 나타내는 벡터는 그 문서를 이루는 단어 벡터들을 모두 더한 것과 같다.

one-hot encoding은 단어의 의미를 전혀 고려하지 않으며 벡터의 길이가 총 단어 수가 되므로 매우 희박(sparse)한 형태가 된다는 문제가 있다. 이를 해결하기 위해 단어의 의미를 고려하여 좀 더 조밀한 차원에 단어를 벡터로 표현하는 것을 단어 임베딩(word embedding)이라고 한다.

아래 그림은 이러한 단어 임베딩의 한 가지 예시이다. 왼쪽 그림을 보면 왕과 여왕, 왕과 남자, 여왕과 여자가 같은 방향에 있다. 의미가 비슷한 단어는 비슷한 방향에 위치하게 된다. 단어 임베딩은 단어의 의미를 효과적으로 표현하기 때문에 one-hot encoding보다 학습 성능을 높일 수 있다. 또한 대량의 데이터로 단어 임베딩을 미리 학습시켜 두면, 문서 분류와 같은 과제에서 더 적은 데이터로도 학습된 임베딩을 사용하여 높은 성능을 낼 수 있다.





단어 임베딩의 종류에는 LSA, Word2Vec, GloVe, FastText 등이 있다.

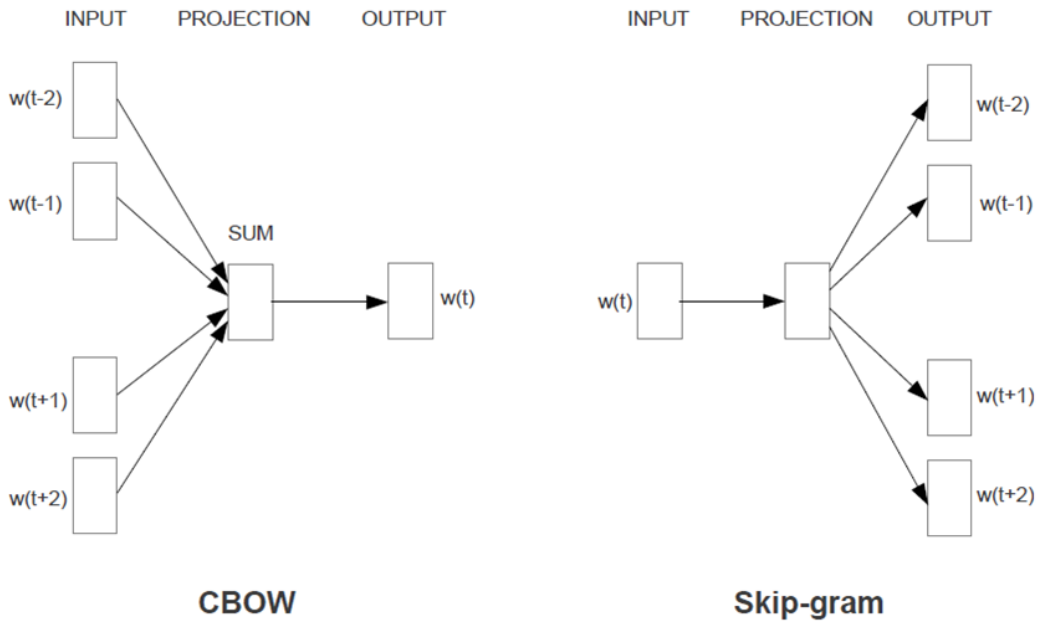
LSA는 단어 문서 행렬을 SVD를 이용해 차원을 줄이는 것이다. TDM을 뒤집어서 문서를 축, 단어를 좌표로 보고 차원을 줄이면 단어 임베딩을 얻을 수 있다. LSA의 적은 데이터로도 잘 작동한다. 하지만 문서 전체의 단어 통계에 바탕을 두기 때문에 단어가 등장하는 세밀한 맥락은 잘 고려하지 못한다.

11.1. Word2Vec

Word2Vec은 2013년에 Mikolov 와 동료들이 제안한 모형이다. Word2Vec은 이전의 NNLM이라는 신경망 기반 언어 모형을 효율적 학습이 가능하도록 고친 모형이다. 언어 모형(language

model)이란 이전에 나온 단어들을 바탕으로 다음 단어를 예측하는 모형을 가리킨다. 분류 등과 같이 별도의 레이블이 없이 텍스트 자체만 있어도 학습이 가능하다. Word2Vec은 이 언어 모형을 학습시킬 때 생기는 부산물을 이용해서 단어 임베딩을 만든다.

Word2Vec에는 CBOW(continuous bag-of-words)와 Skip-Gram 두 가지 방식이 있다. 둘 다 대상 단어와 주변 단어의 관계를 이용한다. 예를 들어 "나는 매일 파이썬을 공부한다"라는 문장에서 '파이썬'를 대상 단어로 하자. 여기에 윈도(window)를 좌우 2단어라고 하면 주변 단어는 "는", "매일", "을", "공부"가 될 것이다. CBOW는 주변단어의 임베딩을 더해서 대상단어를 예측하고, Skip-Gram은 대상 단어의 임베딩으로 주변단어를 예측한다.



Skip-Gram이 CBOW보다 성능이 더 좋은 편이다. 다만 한 번에 여러 단어를 예측해야 하기 때문에 비효율적인 부분이 있다. 최근에는 negative sampling이라는 방법을 사용한다. 두 단어를 입력으로 넣어서 한 윈도에 등장했던 적이 있다면 1, 아니라면 0을 예측하도록 학습시키는 것이다. 위의 예에서 "파이썬"과 "매일"은 한 윈도에 등장했으므로 1, "나"와 공부"는 같은 윈도에 등장하지 않았으므로 0이 된다. 현재는 Word2Vec은 거의 SGNS(Skip-Gram with Negative Sampling)으로만 학습시킨다.

11.2. Word Embedding

구텐베르크 프로젝트에서 그림 형제의 동화를 다운받아 사용한다.

```
import requests
```

```
import re
```

```
res = requests.get('https://www.gutenberg.org/files/2591/2591-0.txt')
```

앞과 뒤에서 불필요한 부분을 완전 제외한다.

```
grimm = res.text[2801:530661]
```

정규표현식을 이용하여 불필요한 단어는 제외한다.

```
grimm = re.sub(r'[^a-zA-Z\. ]', ' ', grimm)
```

`.split('. ')` 을 이용하여 마침표 단위로 자르자.

```
sentences = grimm.split('. ')  # 문장 단위로 자름
```

이번엔 단어(빈칸) 단위로 자르도록 하자.

```
data = [s.split() for s in sentences]
```

단어 단위로 자른 결과를 확인해보자.

```
data[0]   # 첫 번째 문장을 단어 단위로 자른 결과를 확인하자
```

```
['THE',
 'GOLDEN',
 'BIRD',
 'A',
 'certain',
 'king',
 'had',
 'a',
 'beautiful',
 'garden',
 'and',
 'in',
 'the',
 'garden',
 'stood',
 'a',
 'tree',
 'which',
 'bore',
 'golden',
 'apples']
```

11.2.1. gensim

설치

이제 word2vec 를 필요한 `gensim` 패키지를 설치해보자.

```
!conda install -y gensim
```

11.2.2. Word2Vec

`gensim` 패키지에서 `Word2Vec` 을 불러오자.

```
from gensim.models.word2vec import Word2Vec
```

학습

`Word2Vec()` 함수를 통해 word2vec을 실행할 수 있다.

`sg` 옵션에 0을 넘겨주면 CBOW, 1을 넘겨주면 Skip-gram 이다. `size` 로 벡터의 크기를 지정할 수 있다.

```
model = Word2Vec(data,          # 리스트 형태의 데이터
                 sg=1,          # 0: CBOW, 1: Skip-gram
                 size=100,      # 벡터 크기
                 window=3,      # 고려할 앞뒤 폭(앞뒤 3단어)
                 min_count=3,   # 사용할 단어의 최소 빈도(3회 이하 단어 무시)
                 workers=4)     # 동시에 처리할 작업 수(코어 수와 비슷하게 설정)
```

저장 & 불러오기

`.save()` 를 통해 word2vec 모델을 저장할 수 있다.

```
model.save('word2vec.model')
```

저장한 모델을 불러 올 때는 `.load()` 를 사용한다.

```
model = Word2Vec.load('word2vec.model')
```

단어를 벡터로 변환하기

```
model.wv['princess']
```

```
array([ 0.02655056,  0.10885577, -0.3336497 , -0.01923897,  0.09305191,
       -0.27636686, -0.14564317,  0.17176045, -0.09843703, -0.26681355,
       -0.07594544, -0.22842817,  0.23898636,  0.05326742, -0.17338812,
       -0.0346326 ,  0.04283164,  0.16636392, -0.37561128,  0.02862819,
        0.2405934 ,  0.06736574,  0.07213553,  0.07933741,  0.01409239,
        0.06541474,  0.16836987,  0.13678738,  0.22837223, -0.07647555,
       -0.05870688, -0.13160664, -0.03700178, -0.01250235, -0.3585302 ,
       -0.0636082 ,  0.0080068 ,  0.00720702,  0.05650114, -0.02804025,
       -0.00716483,  0.00590491,  0.15767163,  0.12585361, -0.00298565,
       -0.03594063, -0.15925072, -0.10299328,  0.09406696,  0.13813972,
       -0.1106873 , -0.29165223,  0.02585786, -0.12026282, -0.0635815 ,
        0.20486869,  0.03521083, -0.12777852, -0.1194846 , -0.25254437,
       -0.06918804, -0.22082822, -0.06730497, -0.00319947, -0.23822308,
       -0.10826807, -0.26909107,  0.15125225,  0.07503095,  0.07240632,
       -0.11993805, -0.0746782 ,  0.36219984,  0.1250619 , -0.22832122,
        0.00464966,  0.04987318, -0.23297407,  0.01473408,  0.16759492,
       -0.21473846,  0.29151434, -0.00456638, -0.16993526,  0.01871275,
       -0.14834508, -0.03264757,  0.1631111 ,  0.01001021,  0.18118179,
        0.01639039,  0.05196264, -0.11209603, -0.09924992, -0.00561307,
       -0.22174971,  0.16849378,  0.10897917, -0.03661951,  0.1261714 ],
      dtype=float32)
```

유비(analogy)

`wv.similarity()` 에 두 단어를 넘겨주면 코사인 유사도를 구할 수 있다.

```
model.wv.similarity('princess', 'queen')
```

```
0.9874099652998212
```

`wv.most_similar()` 에 단어를 넘겨주면 가장 유사한 단어를 추출할 수 있다.

```
model.wv.most_similar('princess')
```

```
[('boy', 0.9938170313835144),
 ('youth', 0.9926948547363281),
 ('wolf', 0.9895462989807129),
 ('cook', 0.9888262748718262),
 ('fox', 0.9883263111114502),
 ('girl', 0.9882733225822449),
 ('palace', 0.9879066348075867),
 ('prince', 0.9876989126205444),
 ('wedding', 0.9868041276931763),
 ('second', 0.9867663979530334)]
```

`.wv_most_similar()` 에 `positive` 와 `negative` 라는 옵션을 넘겨줄 수 있다.

"woman:princess = man:?"의 유비를 풀어보자.

```
model.wv.most_similar(positive=['man', 'princess'], negative=['woman'])
```

```
[('prince', 0.9825000762939453),
 ('huntsman', 0.9774874448776245),
 ('cook', 0.9766477346420288),
 ('bride', 0.9765883684158325),
 ('mother', 0.9761663675308228),
 ('miller', 0.9751245975494385),
 ('eldest', 0.9750953912734985),
 ('bird', 0.9750479459762573),
 ('fisherman', 0.9750304222106934),
 ('cat', 0.9746482372283936)]
```

11.2.3. gensim으로 학습된 단어 임베딩을 케라스에서 불러오기

```
from keras.models import Sequential
from keras.layers import Embedding
```

```
NUM_WORDS, EMB_DIM = model.wv.vectors.shape
```

gensim 으로 학습된 단어 임베딩 model.wv.vectors 를 케라스의 임베딩 레이어의 가중치로 사용한다.

```
emb = Embedding(input_dim=NUM_WORDS, output_dim=EMB_DIM,
                trainable=False, weights=[model.wv.vectors])
```

```
net = Sequential()
net.add(emb)
```

'princess'의 임베딩된 벡터 값을 확인해보면 gensim 에서와 같다는 것을 확인할 수 있다.

```
i = model.wv.index2word.index('princess')
```

```
net.predict([i])
```

```
array([[[ 0.05186263,  0.09886628, -0.36954704, -0.1151737 ,
         -0.09292138,  0.32287335,  0.02322933,  0.06584648,
          0.10327567,  0.18287872,  0.23039219, -0.15480566,
         -0.00144594,  0.10652337, -0.21365416,  0.24519314,
         -0.01272466, -0.24670881,  0.23654291, -0.05736439,
          0.10232331,  0.02673805, -0.02908026,  0.03745194,
         -0.10675579, -0.16704334,  0.0463514 ,  0.07267679,
          0.21809985, -0.0625468 ,  0.10494982, -0.10991403,
         -0.04019533,  0.02213044, -0.2616629 , -0.02903773,
         -0.00149394, -0.08290774, -0.00744341, -0.19629766,
          0.18949713, -0.2951216 ,  0.16581888,  0.02814515,
         -0.07274172,  0.36760768, -0.21122281, -0.16384546,
         -0.02614794, -0.04692538, -0.07235318,  0.12435664,
          0.20332205,  0.24424402,  0.09038968,  0.14340629,
         -0.11361171,  0.03696093,  0.03736388, -0.28363636,
          0.10169251, -0.25155884, -0.10785993,  0.24412261,
         -0.11825875, -0.2038133 ,  0.23685236, -0.18423462,
         -0.09088275, -0.09017443,  0.27296302, -0.01637266,
         -0.06862711,  0.16498673, -0.1605394 ,  0.09587181,
          0.03009311, -0.07843032, -0.11048649, -0.07389657,
         -0.17780404, -0.1678595 ,  0.07535231, -0.14141686,
         -0.02882383, -0.15537427,  0.09284303, -0.18460383,
         -0.29603457,  0.03373666,  0.06208549, -0.06347001,
          0.05838713, -0.01157669, -0.1447969 , -0.02819149,
         -0.14444825, -0.00113276,  0.07711551,  0.00351177]]],
      dtype=float32)
```

11.2.4. 미리 학습된 임베딩 사용하기

영어

https://code.google.com/archive/p/word2vec 에서 1천억 단어 규모의 구글 뉴스 데이터로 300만개의 단어의 임베딩을 미리 학습시킨 Word2Vec 임베딩을 다운 받을 수 있다. 중간의 `GoogleNews-vectors-negative300.bin.gz` 를 클릭하여 다운 받으면 된다. 다운받아 압축을 풀면 약 3GB 크기의 파일이 생긴다.

이 파일은 `gensim` 을 이용해 불러들일 수 있다.

```
from gensim.models import KeyedVectors
```

파일이 크기 때문에 불러들이는데 시간이 걸린다.

```
word2vec = KeyedVectors.load_word2vec_format('GoogleNews-vectors-negative300.bin', bin
```

앞에서와 마찬가지로 단어의 임베딩은 `word2vec.wv.vectors` 에 저장되어 있다. 이를 케라스 모형에 그대로 넣을 수 있다.

```
word2vec.vectors.shape
```

```
(3000000, 300)
```

한국어

영어 이외의 미리 학습된 임베딩은 https://github.com/Kyubyong/wordvectors 에서 찾을 수 있다. 한국어 Word2Vec은 `Korean (w)` 라는 링크에서 다운 받을 수 있다. `ko.zip` 을 다운받아 압축을 풀면 50MB 가량의 `ko.bin` 파일이 생긴다. 이 파일은 `.load` 로 불러들일 수 있다.

```
import numpy
```

```
kovec = Word2Vec.load('ko.bin')
```

"한국:서울 = 일본:?"의 유비를 풀어보자.

```
new_foods = set()
for food in foods:
    for word, _ in kovec.most_similar(food):
        new_foods.add(word)
foods |= new_foods
```

```
kovec.wv.most_similar(positive=['일본', '서울'], negative=['한국'])
```

```
[('도쿄', 0.49620240926742554),
 ('영등포', 0.4607112407684326),
 ('서울특별시', 0.45662832260131836),
 ('경성', 0.44781729578971863),
 ('아현동', 0.4475313723087311),
 ('경성부', 0.4472092390060425),
 ('세종로', 0.44181060791015625),
 ('혜화동', 0.44022461771965027),
 ('원효로', 0.4394114017486572),
 ('상도동', 0.4373798370361328)]
```

11.3. keras에서 Word2Vec 직접 학습

11.3.1. 데이터 준비

imdb 데이터셋을 불러온다.

```
from keras.datasets import imdb
```

```
(x_train, y_train), (x_test, y_test) = imdb.load_data()
```

단어 번호를 불러온다.

```
word_index = imdb.get_word_index()
```

단어 번호와 단어의 관계를 사전으로 만든다. 1번은 문장의 시작, 2번은 사전에 없는 단어(Out of Vocabulary)로 미리 지정되어 있다.

```
index_word = {idx+3: word for word, idx in word_index.items()}
```

```
index_word[1] = '<START>'
index_word[2] = '<UNKNOWN>'
```

단어 번호로 된 데이터를 단어로 변환해 본다. 실제 데이터 분석에서는 단어로 된 데이터를 단어 번호로 변환해야 한다.

```
' '.join(index_word[i] for i in x_train[0])
```

```
"<START> this film was just brilliant casting location scenery story direction everyone
```

단어의 총 갯수를 변수에 할당한다.

```
NUM_WORDS = max(index_word) + 1
```

11.3.2. 텍스트를 단어 번호로 바꾸기

텍스트를 단어 번호로 바꾸는 방법을 알아보기 위해 먼저 데이터를 텍스트로 역변환하자.

```
texts = []
for data in x_train:
    text = ' '.join(index_word[i] for i in data)
    texts.append(text)
```

```
len(texts)
```

```
25000
```

텍스트를 단어번호로 바꾸는 것은 `Tokenizer` 를 사용한다.

```
from keras.preprocessing.text import Tokenizer
```

`Tokenizer` 를 생성한다.

```
tok = Tokenizer()
```

`fit_on_texts` 를 이용해 `texts` 에 있는 단어들에 번호를 매긴다.

```
tok.fit_on_texts(texts)
```

`texts_to_sequences` 를 이용해 텍스트를 실제로 단어 번호 리스트로 변환한다.

```
new_data = tok.texts_to_sequences(texts)
```

```
new_data[0][:10]
```

```
[28, 11, 19, 13, 41, 526, 968, 1618, 1381, 63]
```

```
x_train[0][:10]
```

```
[1, 14, 22, 16, 43, 530, 973, 1622, 1385, 65]
```

11.3.3. 단어쌍 만들기

```
from keras.preprocessing.sequence import make_sampling_table, skipgrams
```

단어의 수를 구한다.

```
VOCAB_SIZE = len(tok.word_index)
```

단어를 무작위로 추출하면 자주 나오는 단어가 더 많이 나오게 된다. 이를 방지하기 위해 단어를 추출할 확률을 균형을 맞춘 샘플링 표를 만든다.

```
table = make_sampling_table(VOCAB_SIZE)
```

두 단어씩 뽑아 좌우 2단어(`window_size=2`) 안에 들어있는 경우가 있는지 없는지를 확인하여 데이터를 만든다.

```
couples, labels = skipgrams(data,
                            VOCAB_SIZE,
                            window_size=2,
                            sampling_table=table)
```

`couples` 에는 대상단어와 맥락단어 쌍이 들어있고

```
couples[:5]
```

```
[[7585, 2197], [272, 17222], [1310, 53349], [1271, 848], [6139, 77257]]
```

`labels` 에는 윈도 안에 들어있는 경우가 있었으면 1, 아니면 0으로 코딩되어 있다.

```
labels[:5]
```

```
[1, 0, 0, 1, 0]
```

대상 단어는 `word_target` 으로, 맥락 단어는 `word_context` 로 모은다.

```
word_target, word_context = zip(*couples)
```

어레이로 바꾼다.

```
import numpy as np
```

```
word_target = np.array(word_target, dtype="int32")
word_context = np.array(word_context, dtype="int32")
```

11.3.4. Skip-gram 모형

Skipgram 모형은 `Sequential` 대신 각 레이어를 함수처럼 사용하는 Functional API를 사용해야 한다.

```
from keras.layers import Activation, Dot, Embedding, Flatten, Input, Reshape
```

```
from keras.models import Model
```

입력 레이어를 만든다.

```
input_target = Input(shape=(1,))
input_context = Input(shape=(1,))
```

임베딩 레이어를 만든다.

```
emb = Embedding(input_dim=VOCAB_SIZE, output_dim=8)
```

대상 단어를 입력으로 받는 임베딩 레이어 `target` 과 맥락 단어를 입력으로 받는 임베딩 레이어 `context` 를 만든다. 두 레이어는 모두 `emb` 에서 나온 것이기 때문에 파라미터를 공유한다.

```
target = emb(input_target)
context = emb(input_context)
```

두 임베딩의 내적(dot product)를 구한다. 두 벡터 $(a_1, a_2, \ldots, a_n)$과 $(b_1, b_2, \ldots, b_n)$의 내적은 $\sum_i a_i \cdot b_i$이다. 두 벡터가 비슷할 수록 내적은 커지게 된다.

```
dot = Dot(axes=2)([target, context])
```

`(1, 1)` 형태로 되기 때문에 `(1, )` 형태로 바꿔준다.

```
flat = Reshape((1,))(dot)
```

마지막으로 활성화 함수를 적용한다.

```
out = Activation('sigmoid')(flat)
```

이제 입력에서 출력으로 이어지는 모형을 만든다.

```
skipgram = Model(inputs=[input_target, input_context], outputs=out)
```

모형의 요약을 확인하자.

```
skipgram.summary()
```

11.3.5. 훈련

```
from keras.optimizers import Adam
```

```
skipgram.compile(optimizer=Adam(), loss='binary_crossentropy', metrics=['accuracy'])
```

```
skipgram.fit(
    [word_target, word_context],
    labels,
    epochs=30)
```

임베딩 레이어 저장

```
import numpy
```

```
numpy.save('emb.npy', emb.get_weights()[0])
```

11.3.6. 임베딩 레이어 재사용

워드 임베딩을 학습시키는 이유는 그 자체로 목적이 있다기보다 다른 학습에 이를 재사용하여 학습 효율을 높이기 위해서다. 저장한 레이어의 가중치를 불러온다.

```
w = numpy.load('emb.npy')
```

임베딩 레이어를 만든다. 아래에서 `trainable=False` 로 하면 임베딩 레이어는 추가 학습을 하지 않는다.

```
emb_ff = Embedding(input_dim=NUM_WORDS, output_dim=8, input_length=30,
                   weights=[w],   # 레이어 가중치를 저장한 값으로 설정한다
                   trainable=False)
```

11.4. 미리 학습된 언어 모형

Word2Vec은 일종의 언어 모형으로 임베딩을 학습하지만, 사용할 때는 단어마다 벡터 값이 고정되어 있어서 단어가 쓰인 맥락을 고려하지 않는다. 최근에는 언어 모형을 미리 학습시켜 두었다가 모형의 하부에 붙여 사용하는 다양한 방법들이 제안되고 있다.

11.4.1. ULMFiT

ULMFiT(Universal Language Model Fine-tuning)은 해석하자면 "보편 언어 모형 미세 조정"이라는 뜻이다. ULMFiT은 1) 언어 모형 학습 2) 언어 모형 미세 조정 3) 분류기 미세 조정의 3단계로 진행된다.

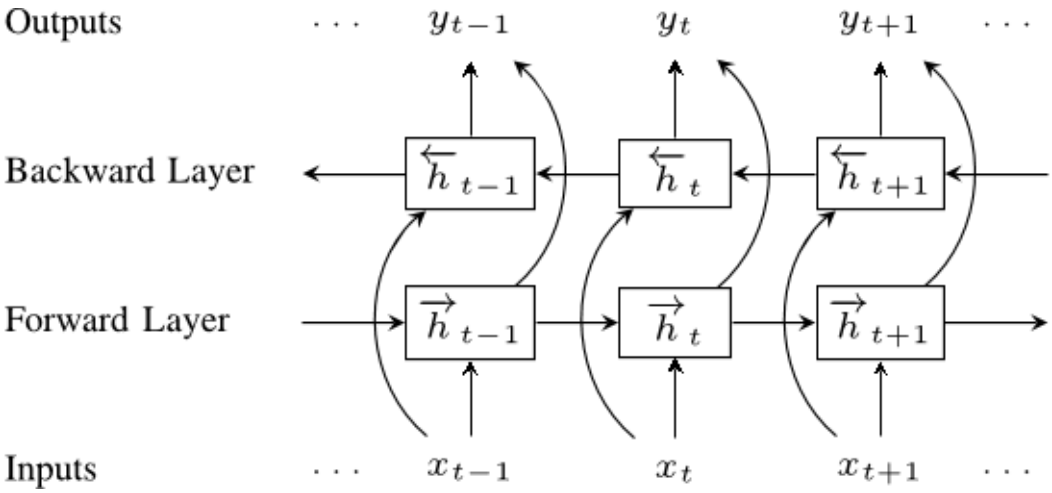
1단계 언어모형 학습은 백과사전이나 뉴스와 같은 광범위한 주제의 텍스트에 언어 모형을 학습시킨다.

2단계 언어모형 미세 조정은 학습된 모형을 제품 리뷰 등 다루고자 하는 주제의 텍스트로 추가 학습시키는 것이다. 이때 레이어마다 학습률을 달리하는 차별적 미세조정(discriminative fine-tuning), 학습률을 빠르게 높였다가 천천히 떨어트리는 기울어진 삼각형 학습률(slanted triangular leraning rate) 등의 테크닉을 사용한다.

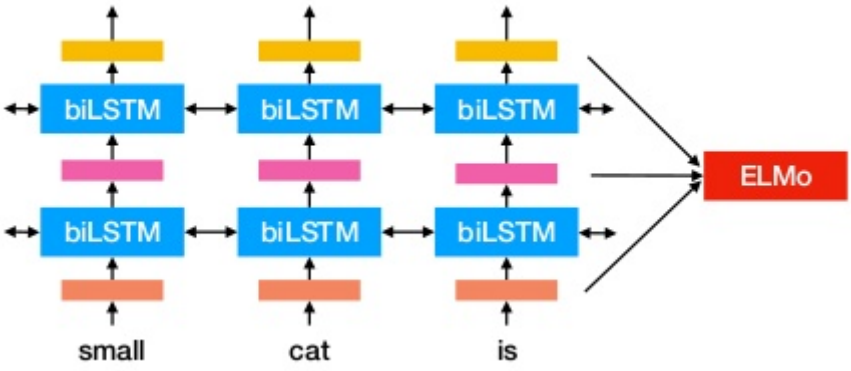
마지막 3단계에서는 미세 조정된 언어 모형 뒤로 리뷰의 긍부정 예측과 같은 특정 과제를 수행하는 분류기를 붙이고 추가 미세 조정을 한다.

11.4.2. ELMo

ELMo(Embeddings from Language Models)는 여러 레이어으로 된 양방향 순환신경망 언어 모형을 사용한다. 아래 그림은 하나의 레이어로 된 양방향 순환신경망인데, 이런 레이어를 여러 층으로 쌓는 것이다.



보통은 개별 과제의 분류기를 언어 모형 뒤에 붙이지만, ELMo는 여러 레이어의 출력값을 모두 사용한다. 왜냐하면 딥러닝에서 한 레이어는 데이터에 가중치를 곱하고 활성화 함수를 거쳐서 새롭게 표상(representation)한다. 표상이란 어떤 실체를 다른 방식으로 표현하는 것을 말한다. 레이어를 겹쳐 쌓으면 레이어마다 다른 방식의 표상을 학습하게 된다. 언어 모형에서 뒷부분의 레이어로 갈 수록 언어 모형에 직접적으로 관련된 표상을 하게 되는데, 과제에 따라서는 뒷부분 레이어보다 앞부분 레이어의 표상이 더 유용할 수도 있다. 우리가 그것을 미리 알 수는 없으므로 모든 레이어의 출력을 개별 과제의 분류기에 입력으로 넣는 것이다.



11.5. ELMo 실습

텐서플로 허브는 텐서플로 모형을 공개하는 저장소이다. ELMo도 허브에 올라와 있으므로 내려받기만 하면 된다. 먼저 허브를 사용하기 위한 라이브러리를 설치한다.

```
!pip install tensorflow-hub
```

허브에서 ELMo 모형을 내려받는다.

```
import tensorflow_hub as hub
```

```
elmo = hub.Module("https://tfhub.dev/google/elmo/1", trainable=True)
```

11.5.1. 케라스 레이어 만들기

ELMo는 텐서플로 모형이므로 케라스의 레이어로 만들어줘야 한다.

```
import tensorflow as tf
```

```
from keras.layers import Lambda
```

케라스의 데이터를 텐서플로에 맞게 넘겨주고 다시 결과를 케라스에 맞게 변환해주는 함수를 만든다.

```
def elmo_embedding(x):
    return elmo(tf.squeeze(tf.cast(x, tf.string)), signature="default", as_dict=True)[
```

Lambda 는 함수를 케라스 레이어로 바꿔준다.

```
elmo_layer = Lambda(elmo_embedding, input_shape=(1,), output_shape=(1024,))
```

11.5.2. 데이터 준비

데이터는 아마존 리뷰 데이터를 사용한다.

```
import pandas as pd
from keras.preprocessing.sequence import pad_sequences
from sklearn.model_selection import train_test_split
```

```
df = pd.read_csv('amazon_cells_labelled.txt', sep='\t', header=None)
```

0번 열을 리스트로 바꾼다.

```
reviews = df[0].tolist()
```

알파벳으로 된 부분만 단어로 뽑아낸다. 최대 30단어까지만 뽑고, 모든 단어는 소문자로 바꾼다.

```
import re
```

```
alphabet = re.compile('[a-zA-Z]+')
```

```
X = []
for review in reviews:
    words = alphabet.findall(review)[:30]
    s = ' '.join(w.lower() for w in words)
    X.append(s)
```

```
import numpy as np
```

```
X = np.array(X, dtype=object)[:, np.newaxis]
```

데이터를 분할한다.

```
X_train, X_test, y_train, y_test = train_test_split(X, df[1], test_size=.2, random_sta
```

11.5.3. 모형

ELMo를 사용한 모형을 만든다.

```
from keras.models import Model
from keras.layers import Dense, Input
```

```
input_layer = Input(shape=(1,), dtype=tf.string)
```

```
emb_layer = elmo_layer(input_layer)
```

```
hidden = Dense(256, activation='relu')(emb_layer)
```

```
out = Dense(1, activation='sigmoid')(hidden)
```

```
model = Model(inputs=[input_layer], outputs=out)
```

```
model.summary()
```

```
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
```

```
model.fit(X_train,
          y_train,
          validation_data=(X_test, y_test),
          epochs=5,
          batch_size=32)
```