10. 순환신경망

10.1. 순차적 데이터

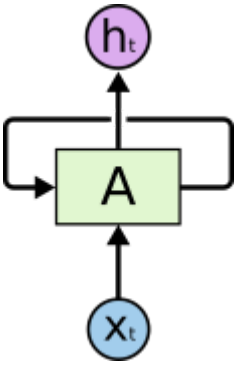
주가나 날씨처럼 시간에 따라 변화하거나, 텍스트나 음악처럼 글자나 음이 순서대로 나타난 정보들이 있다. 이러한 데이터를 **순차적 데이터**(sequential data)라고 한다.

순차적 데이터에서는 대개 순서상 앞이나 뒤에 있는 정보가 서로 영향을 주기도 하고, 주기성이나 경향성을 띄기도 한다. 날씨의 경우 1년을 주기로 추웠다 더워지기를 반복하고, 텍스트의 경우 앞에 나온 말을 보면 뒤에 나올말을 짐작할 수 있다.

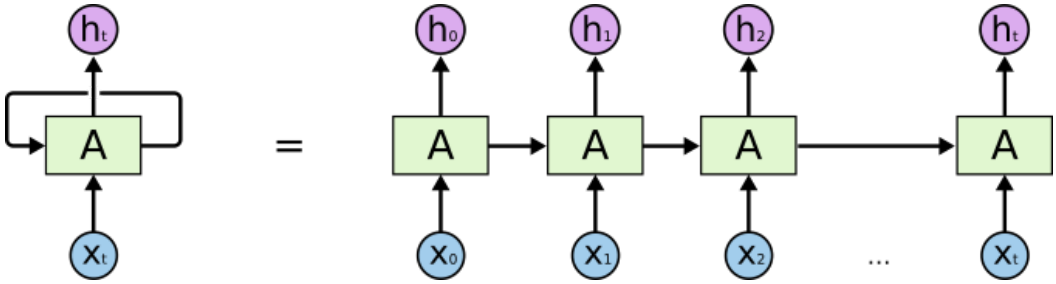
10.2. 순환신경망

순차적 데이터를 다루는 전통적인 방법으로는 자기회귀(autoregression)나 마코브 연쇄(Markov chain)와 같은 방법들이 있다. 딥러닝에서 이러한 특성을 분석할 수 있는 방법으로는 순환신경망(Recurrent Neural Network, 이하 RNN)가 있다.

RNN에는 여러가지 형태가 있으나 가장 대표적인 것은 아래 그림과 같은 형태이다.



입력층 X에서 은닉층 A를 거쳐 출력층 h로 신호가 전달된다는 점에서 RNN은 앞먹임 신경망과 동일한 구조를 가진다. 한 가지 차이는 은닉층에서 자기 자신으로 돌아오는 고리가 있다는 것이다. 즉 첫번째 입력 X_1이 한 번 신경망을 거쳐 나가고 나면, 두번째 입력 X_2가 처리될 때는 X_1의 은닉층 상태가 X_2의 입력과 함께 은닉층으로 들어오게 된다. 이러한 과정을 통해서 앞의 입력이 뒤의 입력에 미치는 영향을 파악할 수 있다. 따라서 RNN을 펼치면 아래와 같은 형태의 네트워크가 된다.



10.3. 장기 의존성의 문제*

RNN에서 은닉층에서 은닉층으로 가는 연결은 같은 신호를 반복해서 전달한다. 즉, 다음과 같은 형태의 식으로 표현된다.

$$A_t = f(A_{t-1}, X_t) = \sigma(wA_{t-1} + vX_t + b)$$

논의를 간단히 하기 위해 입력층과 절편은 제외하고 생각해보자.

$$A_t = \sigma(wA_{t-1})$$

그러면 은닉층의 이전 상태에 대한 다음 상태의 미분은 아래와 같다.

$$\frac{\partial A_t}{\partial A_{t-1}} = w\sigma'(wA_{t-1})$$

만약 은닉층을 여러 층 거칠 경우에 그 미분은 아래와 같은 식이 된다.

$$\begin{aligned}\frac{\partial A_{t+n}}{\partial A_t} &= \frac{\partial A_{t+n}}{\partial A_{t+n-1}} \cdots \frac{A_{t+1}}{A_t} \\ &= \prod_{k=0}^{n-1} \frac{A_{t+k+1}}{A_{t+k}} \\ &= \prod_{k=0}^{n-1} w\sigma'(wA_{t+k}) \\ &= w^n \prod_{k=0}^{n-1} \sigma'(wA_{t+k})\end{aligned}$$

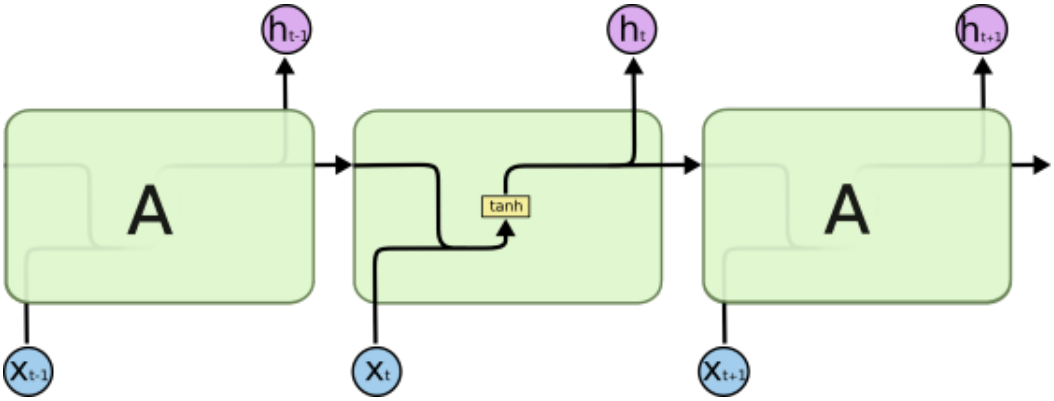
여기서 보면 w^n 때문에 n이 커지면 $w < 1$인 경우 미분이 0으로 수렴하고, $w > 1$인 경우 발산하게 된다. 전자는 사라지는 경사, 후자는 폭발하는 경사(exploding gradient)라고 한다.

사라지는/폭발하는 경사 자체는 앞먹임 신경망에서도 동일한 문제이나 RNN에서는 더 심각한 문제가 된다. 앞먹임 신경망에서는 모형의 깊이가 깊어지면서 생기는 문제지만, RNN에서는 데이터가 길어지기만해도 생기는 문제이기 때문이다.

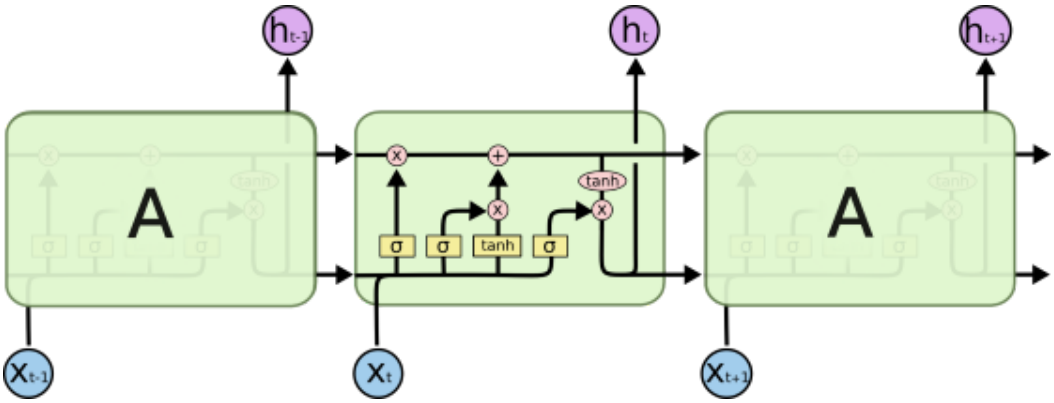
두 입력 사이에 거리가 멀리 떨어진 경우($n \gg 0$)에 존재하는 영향을 장기 의존성(long-term dependency)이라고 한다. 한국어의 텍스트의 경우 주어는 문장의 맨 처음에, 동사는 문장의 맨 끝에 나오므로 문장이 길어지면 주어와 동사 사이에 장기 의존성이 생기게 된다. 그런데 RNN은 두 입력 사이의 거리가 멀면 경사하강법이 잘 작동하지 않아, 장기의존성을 학습하기가 어렵다.

10.4. LSTM

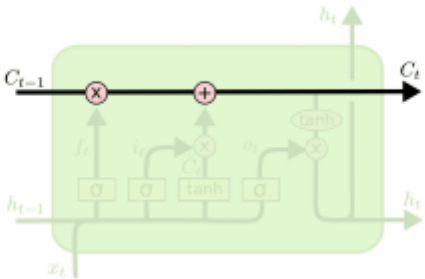
단순한 RNN을 좀 더 자세히 보면 아래와 같은 구조를 가지고 있다. 입력이 이전 은닉층의 상태와 합쳐져 활성화 함수로 들어가고 그 출력이 출력층과 다음 은닉층으로 넘어가는 것이다.



RNN이 장기 의존성을 학습하지 못하는 이유는 사라지는/폭발하는 경사 때문이고, 사라지는/폭발하는 경사는 활성화 함수를 여러 번 반복해서 거치기 때문이다. 그렇다면 은닉층에서 은닉층으로 바로 신호가 전달할 수 있게 하면 어떨까? 이 아이디어에 바탕을 둔 것이 LSTM(Long Short-Term Memory)이다.



매우 복잡하게 보이지만 위의 그림에서 핵심 아이디어는 아래 부분이다.



일단 LSTM에서는 은닉층에서 은닉층으로 전달되는 신호 C와 은닉층에서 출력층으로 전달되는 신호 h가 분리되었다. 그리고 C는 별다른 활성화 함수를 거치지 않고 바로 다음 은닉층으로 전달된다. 따라서 사라지는/폭발하는 경사 문제에서 자유롭게 된다. 대신 망각 게이트(forgetting gate, 위의 그림에서 분홍색 원 안의 X)를 두어 신호를 차단하거나, 입력 게이트(input gate, 분홍색 원 안의 +)를 통해 새로운 신호를 추가한다.

LSTM은 최근에 특히 텍스트, 음성 분석 등에서 각광을 받고 있다. (LSTM 자체는 1997년에 발표된 모형이다.) LSTM은 다음과 같은 문제들에 탁월한 성과를 보여주고 있다.

- 손글씨 인식
- 음성 인식
- 기계 번역

- 이미지 설명 생성
- 문법 분석

10.5. RNN을 이용한 감정 분석

```
import keras
import pandas as pd
```

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

```
df.head()
```

	0	1
0	So there is no way for me to plug it in here i...	0
1	Good case, Excellent value.	1
2	Great for the jawbone.	1
3	Tied to charger for conversations lasting more...	0
4	The mic is great.	1

10.5.1. 토큰화

이제까지는 텍스트를 TDM으로 바꿔서 분석이나 예측을 했다. 순환신경망은 글자나 단어들이 연속해서 나오는 형식의 데이터를 처리하는 모형이므로 입력도 그에 맞게 해줘야 한다.

가장 먼저 할 것은 텍스트를 토큰화하는 것이다. 토큰화는 기존처럼 단어나 형태소 단위로 하기도 하고, 글자 단위로 하기도 한다. 인터넷에 올라오는 글 등은 맞춤법이나 띄어쓰기가 자주 틀리고 신조어도 많기 때문에 단어나 형태소보다 글자 단위로 토큰화하는 것이 유리할 수도 있다.

일단은 단어 단위로 토큰화를 하도록 하자. 케라스의 `Tokenizer` 는 사이킷런의 `CounterVectorizer` 와 비슷하게 텍스트의 토큰화를 해준다. 출력 형태가 TDM이 아닌 토큰의 리스트라는 것이 차이점이다.

```
tok = keras.preprocessing.text.Tokenizer()
```

텍스트가 있는 표의 0번째 열 `df[0]` 을 바탕으로 텍스트에 나타난 단어 종류를 학습한다.

```
tok.fit_on_texts(df[0])
```

학습이라고는 하지만 실제로는 단어마다 고유 번호를 붙인 것이다. 예를 들어 'plug'의 단어 번호는 155번이다.

```
tok.word_index['plug']
```

```
155
```

`단어: 번호` 형태의 사전은 `index_word` 이다.

```
tok.index_word[155]
```

```
'plug'
```

이제 `df[0]` 을 실제로 토큰화를 하고, 텍스트를 단어 번호의 리스트로 변환한다. `CountVectorizer` 와 달리 형태소 분석기를 붙이는 것이 불가능하므로 한국어를 처리할 경우에는 미리 형태소 단위로 띄어쓰기를 해둬야 한다.

```
seq = tok.texts_to_sequences(df[0])
```

첫번째 글은 다음과 같은 리스트로 바뀌어 있다. TDM에서는 33번째 단어가 1개 출현했다는 의미로 33번째 열의 값이 1이 되지만, 여기서는 단순히 33번이 맨 처음에 한 번 나온다.

```
seq[0]
```

```
array([ 33, 117,   5,  53, 214,  11,  47,   8, 155,   4,  19, 337,  19,
         1, 546, 416,   2, 241, 190,   6, 812])
```

단어 번호를 역으로 단어로 바꾸려면 다음과 같이 한다.

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

```
'so there is no way for me to plug it in here in the us unless i go by a converter'
```

10.5.2. 패딩

패딩(padding)이란 텍스트의 앞이나 뒤에 특정한 값을 채워넣어 길이를 맞춰주는 것을 말한다. 만약 모든 텍스트의 길이가 똑같으면 실제로 '순환'시킬 필요 없이 모형을 똑같은 크기로 unrolled시켜서 반복 적용할 수 있기 때문에 더 효율적이다.

먼저 텍스트의 최대 길이를 구한다.

```
MAXLEN = max(len(s) for s in seq)
```

패딩을 통해 길이를 맞춘다. 텍스트가 너무 긴 경우에는 최대 길이보다 더 짧은 길이로 맞추기도 한다. 그렇게 하면 너무 긴 텍스트는 일부가 잘려 나가게 된다.

```
pad = keras.preprocessing.sequence.pad_sequences(seq, MAXLEN)
```

```
pad[0]
```

```
array([  0,   0,   0,   0,   0,   0,   0,   0,   0,  33, 117,   5,  53,
       214,  11,  47,   8, 155,   4,  19, 337,  19,   1, 546, 416,   2,
       241, 190,   6, 812])
```

10.5.3. 데이터 분할

```
from sklearn.model_selection import train_test_split
```

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

10.5.4. 모형 만들기

모형에서 학습할 단어의 갯수는 실제 단어의 갯수에 1을 더한다. 패딩에 사용한 0이 포함되기 때문이다.

```
NUM_WORDS = len(tok.index_word) + 1
```

```
from keras.models import Sequential
from keras.layers import Dense, Embedding, LSTM
```

```
rnn = Sequential()
```

먼저 임베딩 레이어를 넣어준다. 임베딩 레이어는 입력된 단어 번호를 벡터(좌표)로 바꿔준다. 아래의 예에서는 8차원의 벡터로 변환한다. 단어 번호는 단어의 의미와 상관없이 붙은 것이므로 분석에 별로 도움이 되지 않는다. 예를 들어 'phone'은 9번인데, 'smartphone'은 1657번이다. 이 두 단어는 의미가 비슷해서 서로 바꿔쓸 수도 있기 때문에 만약 비슷한 벡터를 가지게 된다면 분석에 도움이 될 것이다. 구체적인 벡터값은 학습을 통해 얻어지게 된다.

`mask_zero=True` 는 0으로 패딩된 값을 마스킹하여 네트워크의 뒤로 전달되지 않게 만든다. 이렇게 하면 인위적으로 패딩된 부분은 학습에 영향을 미치지 않는다.

```
rnn.add(Embedding(input_dim=NUM_WORDS, output_dim=8, input_length=MAXLEN, mask_zero=Tr
```

다음으로 LSTM 레이어를 추가한다. 아래 LSTM은 `16` 개의 노드를 가진다.
`return_sequences=False` 는 가장 마지막 토큰에만 다음 레이어로 출력을 내보내라는 뜻이다.

```
rnn.add(LSTM(16, return_sequences=False))
```

이제 LSTM의 출력을 바탕으로 긍부정을 예측는 `Dense` 레이어를 덧붙인다.

```
rnn.add(Dense(1, activation='sigmoid'))
```

모형의 요약을 보자. `NUM_WORDS` 가 1,879개이고 이들 각각이 8차원 벡터로 바뀌기 때문에 임베딩 레이어는 1,879 * 8 = 15,032개의 파라미터를 갖는다.

LSTM은 이들 8차원 벡터를 16개의 노드로 바꾼다. 여기에 파라미터가 $(8+1) \times 16 = 144$ 개가 들어간다(+1은 바이어스다). 또한 16개의 노드에서 다시 16개의 노드로 순환되는 부분이 있기 때문에 $16 \times 16 = 256$개가 추가되서 400개가 된다. 그리고 이런 게이트가 4개(입력, 출력, 망각, 활성화) 있기 때문에 총 1,600개의 파라미터를 갖는다.

여기까지보면 텍스트의 전체 길이 30은 파라미터의 수에 영향을 주지 않는 것을 알 수 있다. 왜냐하면 순환신경망이기 때문에 동일한 파라미터를 갖는 구조가 30번 반복 적용될 뿐이기 때문이다.

```
rnn.summary()
```

10.5.5. 훈련

훈련시키는 방법은 다른 신경망과 동일하다.

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

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

```
rnn.fit(X_train, y_train, epochs=10)
```

```
<keras.callbacks.History at 0x19f60f38f28>
```

10.5.6. 예측

```
y_rnn = rnn.predict_classes(X_test)
```

```
from sklearn.metrics import accuracy_score
```

```
accuracy_score(y_test, y_rnn)
```

```
0.795
```

10.5.7. 역방향 RNN

단어처리 순서를 앞에서 뒤로 하는 것이 아니라 역방향으로 뒤에서 앞으로도 할 수 있다. 순환 신경망 레이어에 `go_backwards=True` 를 추가해주면 된다.

```
rnn = Sequential()
rnn.add(Embedding(input_dim=NUM_WORDS, output_dim=8, input_length=MAXLEN, mask_zero=Tr
rnn.add(LSTM(16, return_sequences=False, go_backwards=True))
rnn.add(Dense(1, activation='sigmoid'))
```

10.5.8. 양방향 RNN

순방향 RNN과 양방향 RNN을 합치면 양방향 RNN이 된다. `Bidirectional` 을 사용한다.

```
from keras.layers import Bidirectional
```

```
rnn = Sequential()
rnn.add(Embedding(input_dim=NUM_WORDS, output_dim=8, input_length=MAXLEN, mask_zero=Tr
rnn.add(Bidirectional(LSTM(16, return_sequences=False)))
rnn.add(Dense(1, activation='sigmoid'))
```

10.6. 언어 모형

한국어에서 "오늘 밥 먹었다"라는 문장은 "내일 밥 먹었다"라는 문장보다 더 자연스럽다. 확률의 관점에서 이야기하면 앞의 문장이 뒤의 문장보다 더 확률이 높다고 할 수 있다. 이런 문장의 확률을 계산하는 모형을 언어 모형(language model)이라고 한다. 좀 더 자세하게 말하자면 n개의 단어들로 된 문장 $w_1, \ldots, w_n$이 주어졌을 때 이 문장의 확률 분포 $P(\text{w_1, \ldots, w_n})$이다.

언어 모형의 가장 기본적인 형태는 유니그램(unigram) 모형이다. 이 모형은 모든 단어들이 독립적으로 나타난다고 가정한다. 따라서 $P(\text{w_1, \ldots, w_n}) = P(w_1) \times \ldots \times P(w_n)$과 같다. 즉, "오늘 밥 먹었다"라는 문장이 나타날 확률은 '오늘'의 확률, '밥'의 확률, '먹었다'의 확률을 단순히 곱한 것과 같다. 이 모형은 어순이나 맥락을 전혀 고려하지 않는다.

이를 좀 더 확장하면 앞 단어에 따라 다음 단어의 조건부확률을 추정하는 방식의 바이그램(bigram)이 있고, 앞의 2단어를 고려하면 트라이그램(trigram), 3단어를 고려하면 쿼드그램(quadgram) 등등이 된다. 이들을 통틀어서 N-그램이라고 한다.

순환신경망을 이용하면 N-그램보다 더 자연스럽게 언어모형을 만들 수 있다. 여기서는 아마존 리뷰를 바탕으로 언어 모형을 만들어보도록 하자.

아마존 리뷰 데이터에서 첫번째 글을 보면 문장의 시작에는 'So'가 나오고, 'So' 다음에는 'there'이 나오며, 'So there' 다음에는 'is'가 나온다. 이러한 관계를 순환신경망에 학습시키면 된다.

```
df.iloc[0, 0]
```

```
'So there is no way for me to plug it in here in the US unless I go by a converter.'
```

텍스트의 시작과 끝을 나타내는 단어를 사전에 추가해준다.

```
tok.word_index['<START>'] = start = len(tok.word_index) + 1
tok.index_word[start] = '<START>'

tok.word_index['<END>'] = end = len(tok.word_index) + 1
tok.index_word[end] = '<END>'
```

모든 텍스트의 앞과 뒤에 시작과 끝 표시를 붙여 `prev_seq` 를 만들고, 끝 표시만 붙은 `next_seq` 를 만든다. 이렇게 하면 `prev_seq` 와 `next_seq` 는 한 단어씩 어긋나게 된다. 순환 신경망에 `prev_seq` 를 입력으로, `next_seq` 를 출력으로 넣어줄 것이다.

```
prev_seq = []
next_seq = []
for s in seq:
    prev_seq.append([start] + s + [end])
    next_seq.append(s + [end])
```

첫 리뷰를 확인해보면 다음과 같은 형태가 된다.

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

```
'<START> so there is no way for me to plug it in here in the us unless i go by a conver
```

10.6.1. 패딩

텍스트의 최대 길이를 구한다.

```
MAXLEN = max(len(s) for s in prev_seq)
```

패딩을 하는데 이전과 달리 뒤에 0을 넣어 채워준다.

```
from keras.preprocessing.sequence import pad_sequences
```

```
prev_pad = pad_sequences(prev_seq, MAXLEN, padding='post')
next_pad = pad_sequences(next_seq, MAXLEN, padding='post')
```

```
prev_pad[0]
```

```
array([1879,   33,  117,    5,   53,  214,   11,   47,    8,  155,    4,
         19,  337,   19,    1,  546,  416,    2,  241,  190,    6,  812,
       1880,    0,    0,    0,    0,    0,    0,    0,    0,    0])
```

```
next_pad[0]
```

```
array([  33,  117,    5,   53,  214,   11,   47,    8,  155,    4,   19,
        337,   19,    1,  546,  416,    2,  241,  190,    6,  812, 1880,
          0,    0,    0,    0,    0,    0,    0,    0,    0,    0])
```

10.6.2. 데이터 분할

```
from sklearn.model_selection import train_test_split
```

```
x_train, x_test, y_train, y_test = train_test_split(prev_pad, next_pad, test_size=.2,
```

10.6.3. 모형 만들기

감성분석은 텍스트 전체에 대해 마지막에 하나의 긍부정만 예측하면 되었지만, 언어모형은 모든 단어에 대해 다음 단어를 예측해야 한다.

```
NUM_WORDS = len(tok.index_word) + 1
```

```
from keras.models import Sequential
from keras.layers import Dense, Embedding, LSTM, TimeDistributed
```

```
rnn = Sequential()
rnn.add(Embedding(input_dim=NUM_WORDS, output_dim=8, input_length=MAXLEN, mask_zero=Tr
```

`return_sequences=True` 로 모든 입력에 대해 출력을 내놓게 한다.

```
rnn.add(LSTM(16, return_sequences=True))
```

이제 LSTM은 입력된 단어 수만큼 출력을 반복해서 내놓게 된다. 각각의 출력에서 `Dense` 레이어는 다음에 나올 단어를 예측한다. 출력마다 `Dense` 를 반복해서 덧붙이고자 하기 때문에 `TimeDistributed` 를 사용한다.

```
rnn.add(TimeDistributed(Dense(NUM_WORDS, activation='softmax')))
```

모형의 요약을 보면 마지막 레이어의 형태가 `(None, 1879)` 가 아닌 `(None, 32, 1879)` 이다. 입력이 최대 32개이므로 출력도 최대 32개인데 `Dense` 가 반복되기 때문에 형태가 달라진 것이다.

```
rnn.summary()
```

10.6.4. 출력의 형태 맞추기

```
y_train.shape
```

```
(800, 32)
```

출력의 형태가 `(None, 32, 1881)` 인데 데이터는 `(800, 32)` 의 형태이므로 차원이 맞지 않는다. 끝에 1차원을 덧붙여서 형태를 맞춰준다.

```
y_train_dims = numpy.expand_dims(y_train, 2)
```

```
y_train_dims.shape
```

```
(800, 32, 1)
```

10.6.5. 훈련

이제 학습을 시킨다.

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

```
rnn.compile(optimizer=Adam(lr=.1), loss='sparse_categorical_crossentropy', metrics=['a
```

```
rnn.fit(x_train, y_train_dims, epochs=10)
```

```
<keras.callbacks.History at 0x14912613dd8>
```

```
y_train.shape
```

```
(800, 32)
```

10.6.6. 문장 생성

언어 모형을 이용해 문장의 다음 단어를 예측해보자. 예를 들어 첫번째 리뷰의 앞 10단어는 다음과 같다.

```
[tok.index_word[i] for i in prev_seq[0][:10]]
```

```
['<START>', 'so', 'there', 'is', 'no', 'way', 'for', 'me', 'to', 'plug']
```

이어서 나올 단어는 'it'이다.

```
i = prev_seq[0][10]
tok.index_word[i]
```

```
'it'
```

이제 RNN으로 예측을 해보자.

```
new_sentence = [prev_seq[0][:10]]
```

패딩을 하고

```
new_pad = pad_sequences(new_sentence, MAXLEN, padding='post')
```

예측을 한다.

```
next_words = rnn.predict(new_pad)
```

1개의 텍스트에 대해 32단어 길이로 1881종의 단어에 대한 예측이 나왔다.

```
next_words.shape
```

```
(1, 32, 1881)
```

가장 확률이 높은 단어는 4번이다.

```
next_words[0, 10].argmax()
```

```
4
```

4번은 'it'이므로 정확히 예측했다. 이제 완전히 새로운 문장을 만들어보자.

```
new_sentence = [[start]]
```

```
new_pad = pad_sequences(new_sentence, MAXLEN, padding='post')
```

```
next_words = rnn.predict(new_pad)
```

```
next_words[0, 0].argmax()
```

```
2
```

2번 단어가 예측되었다. 이 단어를 `new_sentence` 에 추가하고 다시 위의 과정을 반복하자.

```
new_pad[0, 1] = 2
next_words = rnn.predict(new_pad)
next_words.argmax(axis=2)
```

```
array([[ 2, 22, 22, 22, 22, 22, 22, 22, 22, 22, 22, 22, 22, 22, 22, 22,
        22, 22, 22, 22, 22, 22, 22, 22, 22, 22, 22, 22, 22, 22, 22, 22]],
      dtype=int64)
```

다음으로 22번 단어가 예측되었다. 이를 처음부터 끝까지 반복하게 하면 다음과 같다.

```
new_sentence = [[start]]
new_pad = pad_sequences(new_sentence, MAXLEN, padding='post')

for i in range(MAXLEN - 1):
    next_words = rnn.predict(new_pad) # 예측
    word = next_words[0, i].argmax()  # 가장 확률이 높은 단어 선정
    print(tok.index_word[word])       # 단어 출력
    new_pad[0, i + 1] = word          # 선정 단어를 추가
    if word == end:                   # 문장이 끝나면 중단
        break
```

위의 방식으로 하면 매번 같은 문장이 만들어지므로 다양성이 부족하다. 확률이 가장 높은 단어를 선택하는 대신, 단어를 확률에 따라 무작위로 추출하게 하자.

```
import numpy.random
```

```
new_sentence = [[start]]
new_pad = pad_sequences(new_sentence, MAXLEN, padding='post')

for i in range(MAXLEN - 1):
    next_words = rnn.predict(new_pad)

    # 확률에 따라 단어를 무작위로 추출
    word = numpy.random.choice(NUM_WORDS, p=next_words[0, i])

    print(tok.index_word[word])
    new_pad[0, i + 1] = word
    if word == end:
        break
```