ch17.
딥러닝 자연어처리

17장 딥러닝을 이용한 자연어 처리

1 | 텍스트의 토큰화

2 | 단어의 원-핫 인코딩

3 | 단어 임베딩

4 | 텍스트를 읽고 긍정, 부정 예측하기

딥러닝을 이용한 자연어 처리

- 인공지능 비서 서비스를 누구나 사용하는 시대가 올 것

- 자연어 처리(NLP, Natural Language Processing)
  - 음성이나 텍스트를 컴퓨터가 인식하고 처리하는 것

- AI 스피커
  - 애플 시리, 구글 어시스턴트, 아마존 알렉사, SK Nugu, 네이버 클로바

- 딥러닝이 등장하면서 자연어처리 연구가 활발해짐
  (대용량 데이터를 학습할 수 있는 딥러닝의 속성 때문)

- 컴퓨터는 수치 데이터만 이해할 수 있기 때문에, 자연어처리는 텍스트 전처리 과정이 필수

텍스트의 토큰화

- 먼저 해야 할 일은 텍스트(문장)를 잘게 나누는 것
- 토큰(token) : 텍스트(문장)를 단어별, 문장별, 형태소별로 나눌 수 있는데, 나누어져서 의미가 있는 단위
- 토큰화(tokenization) : 입력된 텍스트를 잘게 나누는 과정

```
# 전처리할 텍스트를 정합니다.
text = '해보지 않으면 해낼 수 없다'
```

- 케라스 text 모듈의 text_to_word_sequence() 함수를 사용하면 문장의 단어 단위 토큰화가 가능
- 해당 함수를 불러와 전처리할 텍스트를 지정한 후, 다음과 같이 토큰화함

```
#주어진 문장을 '단어'로 토큰화 하기
#케라스의 텍스트 전처리와 관련한 함수중 text_to_word_sequence 함수를 불러 옵니다.
from tensorflow.keras.preprocessing.text import text_to_word_sequence

# 전처리할 텍스트를 정합니다.
text = '해보지 않으면 해낼 수 없다'

# 해당 텍스트를 토큰화 합니다.
result = text_to_word_sequence(text)
print("\n원문:\n", text)
print("\n토큰화:\n", result)
```

```
원문:
 해보지 않으면 해낼 수 없다

토큰화:
 ['해보지', '않으면', '해낼', '수', '없다']
```

텍스트의 토큰화 : Bag-of-Words

- 문장을 토큰화하면 이를 이용해 여러 가지를 할 수 있음
  - 단어 빈도 확인
  - 중요 단어 파악

- Bag-of-Words라는 방법이 이러한 전처리를 일컫는 말인데, '단어의 가방(bag of words)'이라는 뜻
  → 같은 단어끼리 따로 따로 가방에 담은 뒤 각 가방에 몇 개의 단어가 들어있는지를 세는 기법

```
#단어 빈도수 세기
#전처리 하려는 세개의 문장을 정합니다.
docs = ['먼저 텍스트의 각 단어를 나누어 토큰화 합니다.',
        '텍스트의 단어로 토큰화 해야 딥러닝에서 인식됩니다.',
        '토큰화 한 결과는 딥러닝에서 사용 할 수 있습니다.',
        ]
```

텍스트의 토큰화 : Bag-of-Words

```
# 토큰화 함수를 이용해 전처리 하는 과정입니다.
token = Tokenizer()                  # 토큰화 함수 지정
token.fit_on_texts(docs)            # 토큰화 함수에 문장 적용

#단어의 빈도수를 계산한 결과를 각 옵션에 맞추어 출력합니다.

print("\n단어 카운트:\n", token.word_counts)
#Tokenizer()의 word_counts 함수는 순서를 기억하는 OrderedDict클래스를 사용합니다.

#출력되는 순서는 랜덤입니다.
print("\n문장 카운트: ", token.document_count)
print("\n각 단어가 몇개의 문장에 포함되어 있는가:\n", token.word_docs)
print("\n각 단어에 매겨진 인덱스 값:\n",  token.word_index)
```

```
단어 카운트:
 OrderedDict([('먼저', 1), ('텍스트의', 2), ('각', 1), ('단어를', 1), ('나누어', 1), ('토큰화', 3), ('합니다', 1), ('단어로', 1),
('해야', 1), ('딥러닝에서', 2), ('인식됩니다', 1), ('한', 1), ('결과는', 1), ('사용', 1), ('할', 1), ('수', 1), ('있습니다', 1)])

문장 카운트:  3

각 단어가 몇개의 문장에 포함되어 있는가:
 defaultdict(<class 'int'>, {'먼저': 1, '합니다': 1, '각': 1, '텍스트의': 2, '나누어': 1, '토큰화': 3, '단어를': 1, '해야': 1,
'딥러닝에서': 2, '단어로': 1, '인식됩니다': 1, '수': 1, '사용': 1, '결과는': 1, '있습니다': 1, '할': 1, '한': 1})

각 단어에 매겨진 인덱스 값:
 {'토큰화': 1, '텍스트의': 2, '딥러닝에서': 3, '먼저': 4, '각': 5, '단어를': 6, '나누어': 7, '합니다': 8, '단어로': 9, '해야': 10,
'인식됩니다': 11, '한': 12, '결과는': 13, '사용': 14, '할': 15, '수': 16, '있습니다': 17}
```

문장의 원-핫 인코딩(one-hotencoding)

- 단어가 문장의 다른 요소와 어떤 관계를 가지고 있는지를 알아보는 방법이 필요함
- 이러한 기법 중에서 가장 기본적인 방법인 원-핫 인코딩(one-hotencoding)을 알아보자
- 다음과 같은 문장이 있다

'오랫동안 꿈꾸는 이는 그 꿈을 닮아간다'

- 벡터 공간으로 바꾸면,

```
(0인덱스) 오랫동안  꿈꾸는  이는  그  꿈을  닮아간다
    ⋮        ⋮       ⋮     ⋮    ⋮   ⋮      ⋮
[   0        0       0     0    0   0      0   ]
```

- 각 단어가 배열 내에서 해당하는 위치를 1로 바꿔주면 -> 벡터화 작업

```
오랫동안 =      [0100000]
꿈꾸는   =      [0010000]
이는     =      [0001000]
그       =      [0000100]
꿈을     =      [0000010]
닮아간다 =      [0000001]
```

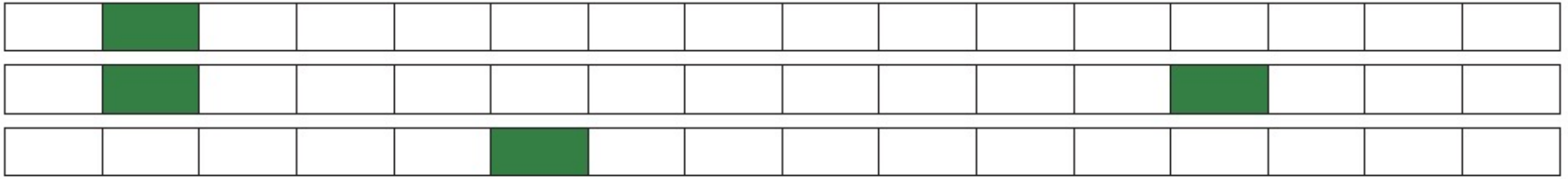
문장의 원-핫 인코딩(one-hotencoding)

```
# 문장의 원-핫 인코딩
from tensorflow.keras.preprocessing.text import Tokenizer

text = "오랫동안 꿈꾸는 자는 그 꿈을 닮아간다"

token = Tokenizer()
token.fit_on_texts([text])
print('문장의 토큰화:', token.word_index)

# 각 단어를 숫자화
x = token.texts_to_sequences([text])
print('문장의 숫자화: ', x)

# 원-핫 인코딩 방식으로 표현하면
from keras.utils import to_categorical

word_size = len(token.word_index) + 1
x = to_categorical(x, num_classes=word_size)
print('문장의 원-핫 인코딩:\n', x)
```

```
문장의 토큰화: {'오랫동안': 1, '꿈꾸는': 2, '자는': 3, '그': 4, '꿈을': 5, '닮아간다': 6}
문장의 숫자화:  [[1, 2, 3, 4, 5, 6]]
문장의 원-핫 인코딩:
 [[[0. 1. 0. 0. 0. 0. 0.]
  [0. 0. 1. 0. 0. 0. 0.]
  [0. 0. 0. 1. 0. 0. 0.]
  [0. 0. 0. 0. 1. 0. 0.]
  [0. 0. 0. 0. 0. 1. 0.]
  [0. 0. 0. 0. 0. 0. 1.]]]
```

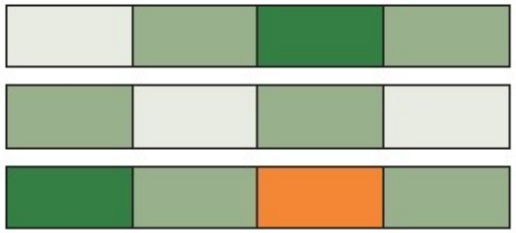
워드 임베딩(word embedding)

- 원-핫 인코딩을 그대로 사용하면 벡터의 길이가 너무 길어진다는 단점이 있음



▲ 원-핫 인코딩

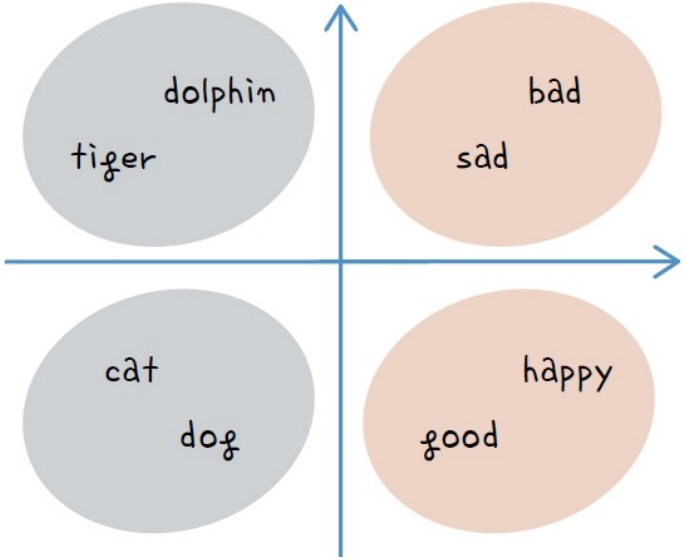
- 공간적 낭비를 해결하기 위해 등장한 것이 단어 임베딩(word embedding)이라는 방법
- 단어 임베딩은 주어진 배열을 정해진 길이로 압축시킴



▲ 단어 임베딩

워드 임베딩(word embedding) - Embedding( ) 함수

- 워드 임베딩으로 얻은 결과는 밀집된 정보를 가지고 있음
- 워드 임베딩을 이용해서 각 단어 간의 유사도를 계산할 수 있음



- 단어 간 유사도 계산은 오차 역전파를 이용
  적절한 크기로 배열을 바꾸어 주기 위해 최적의 유사도를 계산

- Embedding( ) 함수를 사용하면 워드 임베딩을 간단히 해낼 수 있음
- Embedding( ) 함수는 최소 2개의 매개변수를 필요로 하는데, 바로 '입력'과 '출력'의 크기
- Embedding(16, 4) : 입력될 총 단어 수는 16, 임베딩 후 출력되는 벡터 크기는 4
- Embedding(16, 4, input_length=2) : 총 입력되는 단어 수는 16개, 임베딩 후 출력되는 벡터 크기는 4,
  단어를 매번 2개씩 집어 넣겠다는 뜻

패딩(padding)

- 패딩(padding) : 문장의 단어 수를 똑같이 맞춰 주는 작업
- pad_sequnce( ) 함수 : 원하는 길이보다 짧은 부분은 숫자 0을 넣어서 채워주고, 긴 데이터는 잘라줌

```
# 패딩, 서로 다른 길이의 데이터를 4로 맞추어 줍니다.
padded_x = pad_sequences(x, 4)
```

```
[[ 0  0  1  2]
 [ 0  0  0  3]
 [ 4  5  6  7]
 [ 0  8  9 10]
 [ 0 11 12 13]
 [ 0  0  0 14]
 [ 0  0  0 15]
 [ 0  0 16 17]
 [ 0  0 18 19]
 [ 0  0  0 20]]
```

실습 : 영화 리뷰 긍정/부정 예측하기

예제 소스: run_project/17_NLP.ipynb

```
import numpy
import tensorflow as tf
from numpy import array
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense,Flatten,Embedding

#주어진 문장을 '단어'로 토큰화 하기
#케라스의 텍스트 전처리와 관련한 함수중 text_to_word_sequence 함수를 불러 옵니다.
from tensorflow.keras.preprocessing.text import text_to_word_sequence

# 전처리할 텍스트를 정합니다.
text = '해보지 않으면 해낼 수 없다'

# 해당 텍스트를 토큰화 합니다.
result = text_to_word_sequence(text)
print("\n원문:\n", text)
print("\n토큰화:\n", result)

#단어 빈도수 세기
#전처리 하려는 세개의 문장을 정합니다.
docs = ['먼저 텍스트의 각 단어를 나누어 토큰화 합니다.',
        '텍스트의 단어로 토큰화 해야 딥러닝에서 인식됩니다.',
        '토큰화 한 결과는 딥러닝에서 사용 할 수 있습니다.',
        ]

# 토큰화 함수를 이용해 전처리 하는 과정입니다.
token = Tokenizer()              # 토큰화 함수 지정
token.fit_on_texts(docs)         # 토큰화 함수에 문장 적용

#단어의 빈도수를 계산한 결과를 각 옵션에 맞추어 출력합니다.

print("\n단어 카운트:\n", token.word_counts)
#Tokenizer()의 word_counts 함수는 순서를 기억하는 OrderedDict클래스를 사용합니다.

#출력되는 순서는 랜덤입니다.
print("\n문장 카운트: ", token.document_count)
print("\n각 단어가 몇개의 문장에 포함되어 있는가:\n", token.word_docs)
print("\n각 단어에 매겨진 인덱스 값:\n",  token.word_index)
```

```
# 텍스트 리뷰 10개 샘플을 지정합니다.
docs = ["너무 재밌네요","최고예요","참 잘 만든 영화예요",
        "추천하고 싶은 영화입니다","한번 더 보고싶네요",
        "글쎄요","별로예요","생각보다 지루하네요","연기가 어색해요","재미없어요"]

# 긍정 리뷰는 1, 부정 리뷰는 0으로 클래스를 지정합니다.
classes = array([1,1,1,1,1,0,0,0,0,0])

# 토큰화
token = Tokenizer()
token.fit_on_texts(docs)
print(token.word_index)
x = token.texts_to_sequences(docs)
print("\n리뷰 텍스트, 토큰화 결과:\n",  x)

# 패딩, 서로 다른 길이의 데이터를 4로 맞추어 줍니다.
padded_x = pad_sequences(x, 4)
print("\n패딩 결과:\n", padded_x)

#딥러닝 모델
print("\n딥러닝 모델 시작:")

#임베딩에 입력될 단어의 수를 지정합니다.
word_size = len(token.word_index) +1

#단어 임베딩을 포함하여 딥러닝 모델을 만들고 결과를 출력합니다.
model = Sequential()
model.add(Embedding(word_size, 8, input_length=4))
model.add(Flatten())
model.add(Dense(1, activation='sigmoid'))
model.compile(optimizer='adam',
              loss='binary_crossentropy', metrics=['accuracy'])
model.fit(padded_x, classes, epochs=20)
print("\n Accuracy: %.4f" % (model.evaluate(padded_x, classes)[1]))
```

```
Epoch 19/20
1/1 [==============================] - 0s 602us/step - loss: 0.6587 - accuracy: 0.9000
Epoch 20/20
1/1 [==============================] - 0s 585us/step - loss: 0.6566 - accuracy: 0.9000
1/1 [==============================] - 0s 765us/step - loss: 0.6546 - accuracy: 0.9000

 Accuracy: 0.9000
```

10개의 리뷰 샘플 중 긍정/부정 9개를 맞춘다

ch18.
시퀀스배열_순환신경망 (RNN, LSTM)

18장 시퀀스 배열로 다루는 순환 신경망(RNN)

1 | LSTM을 이용한 로이터 뉴스 카테고리 분류하기

2 | LSTM과 CNN의 조합을 이용한 영화 리뷰 분류하기

순환신경망
(RNN)

순환 신경망(RNN, Recurrent Neural Network)

- 순환 신경망(RNN, Recurrent Neural Network)
  - 연속된 데이터가 순서대로 입력되었을 때 앞서 입력받은 데이터를 잠시 기억해 놓는 방법
  - 기억된 데이터 당 중요도 가중치를 주면서 다음 데이터로 넘어감
  - 모든 입력 값에 이 작업을 순서대로 실행. 다음 층으로 넘어가기 전 같은 층을 맴도는 것처럼 보임

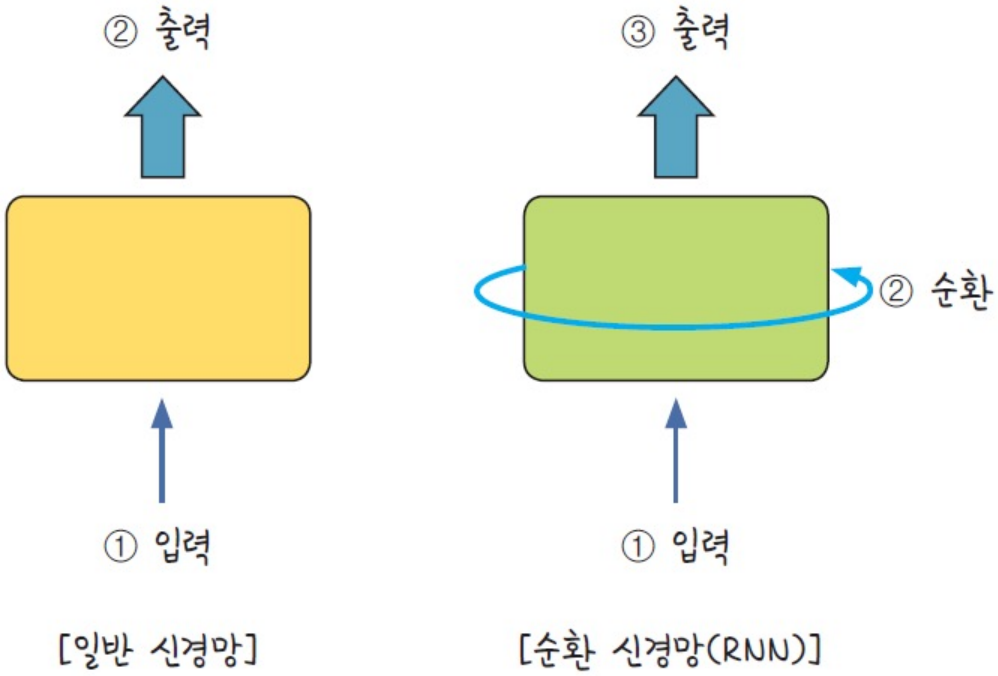


그림 18-1 일반 신경망과 순환 신경망(RNN)의 차이

순환 신경망 예시

- 인공지능 비서에게 "오늘 주가가 몇이야? 라고 묻는다면,

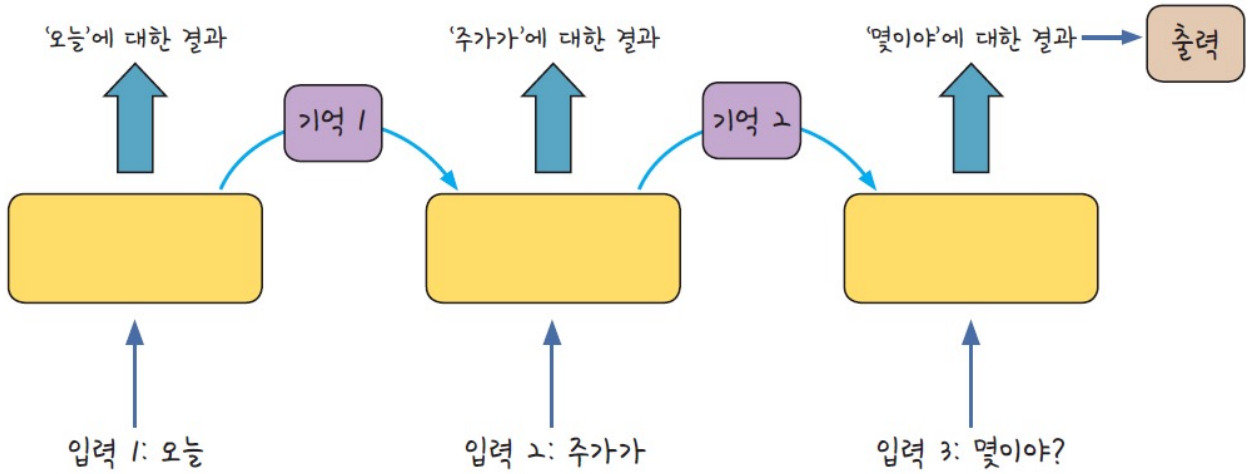


그림 18-2 "오늘 주가가 몇이야?"를 RNN이 처리하는 방식

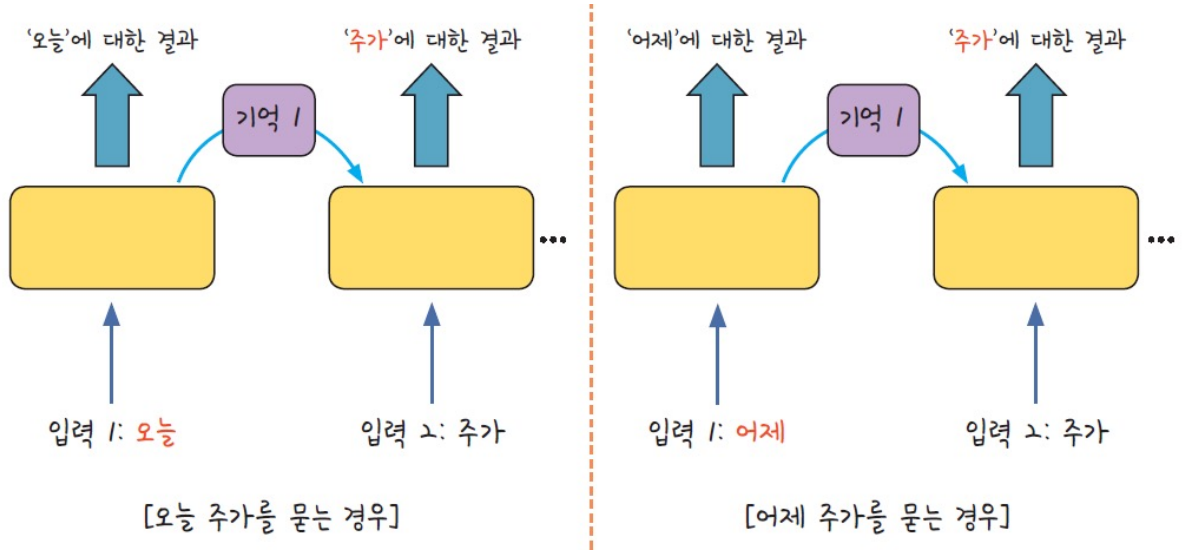


그림 18-3 RNN을 사용하는 이유 "오늘 주가"와 "어제 주가"는 다르다.

LSTM

RNN 개선 : LSTM

- LSTM(Long Short Term Memory) : RNN의 기울기 소실 문제 보완을 위해 나온 방법

  한 층에서 반복되기 직전에 다음 층으로 기억된 값을 넘길지 관리하는 단계를 하나 더 추가하는 것

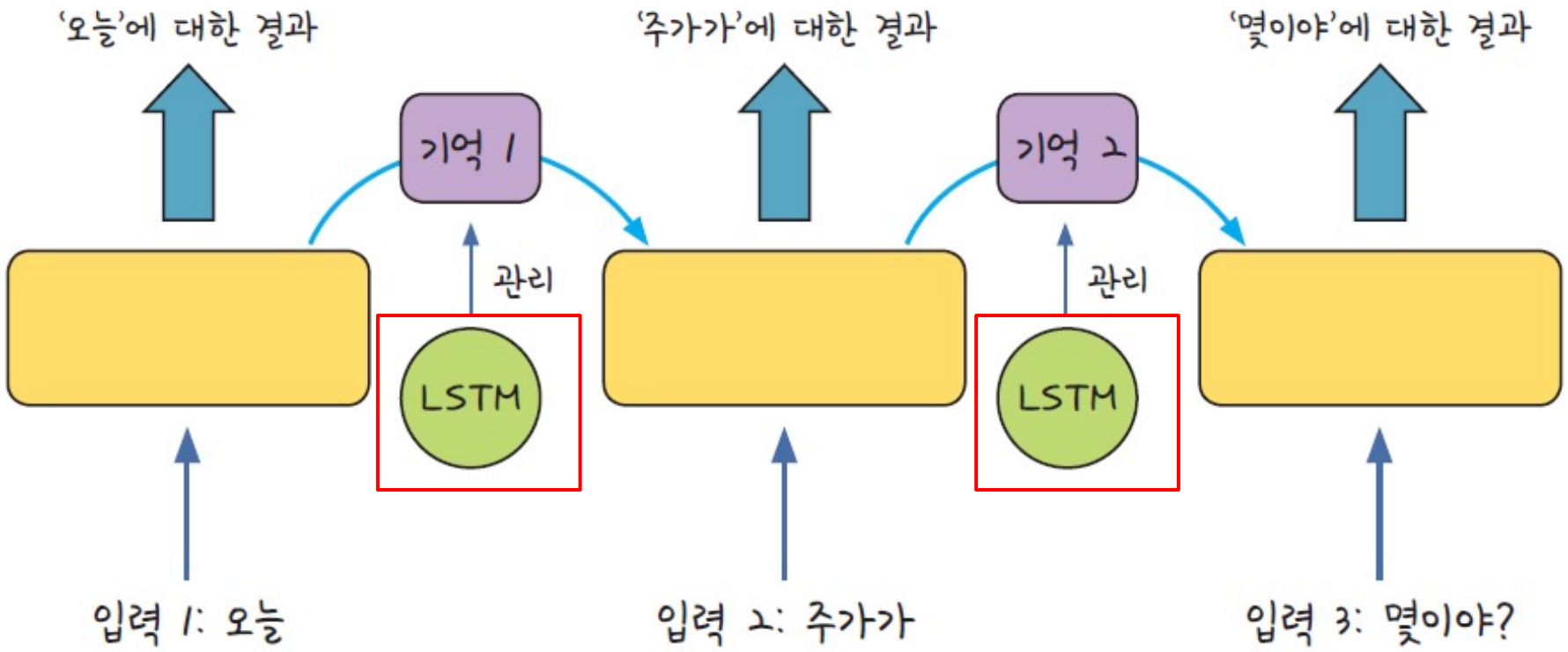
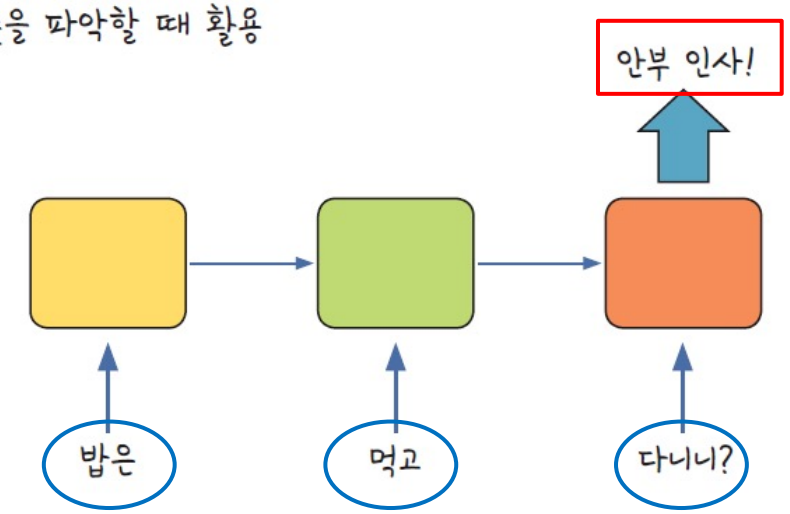


그림 18-4 LSTM은 기억 값의 가중치를 관리하는 장치

RNN(LSTM) 방식의 사용 상황

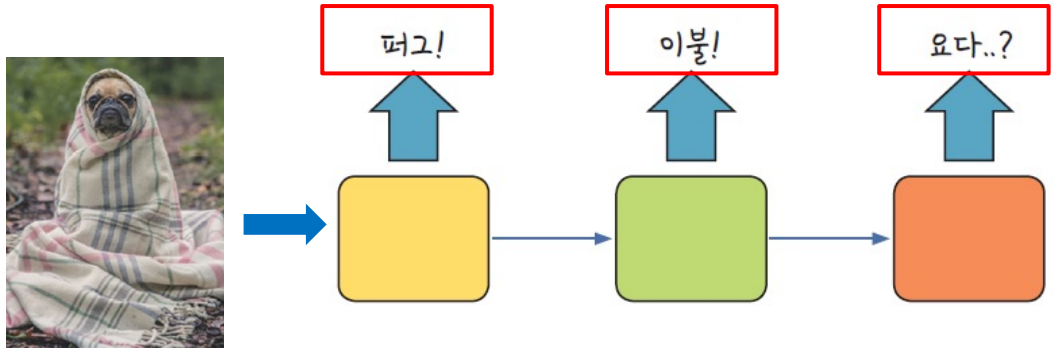
① 다수 입력 단일 출력

예: 문장을 읽고 뜻을 파악할 때 활용



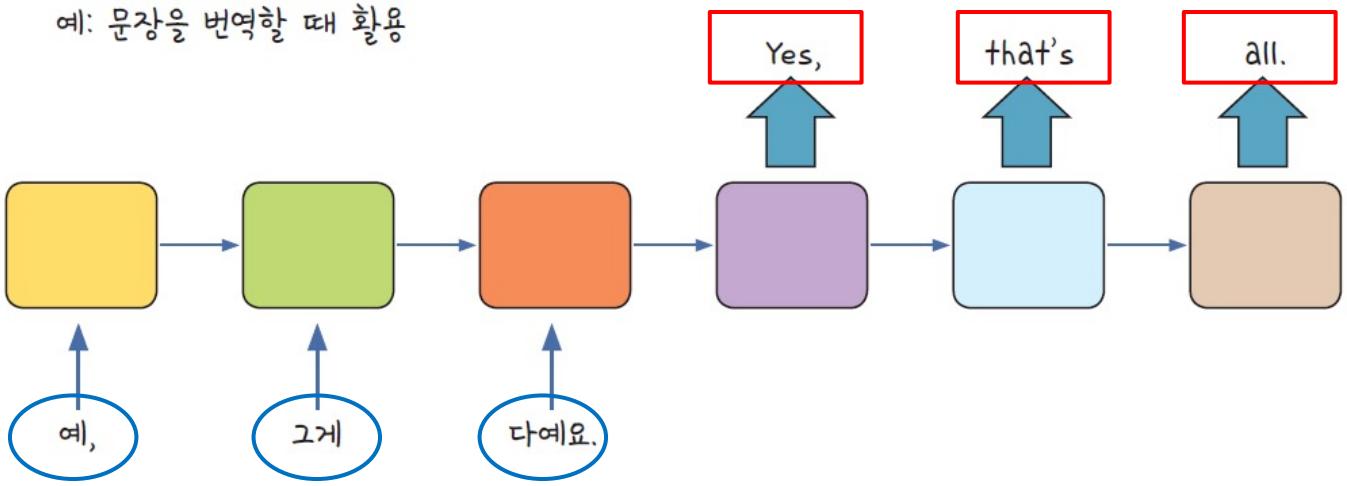
② 단일 입력 다수 출력

예: 사진의 캡션을 만들 때 활용



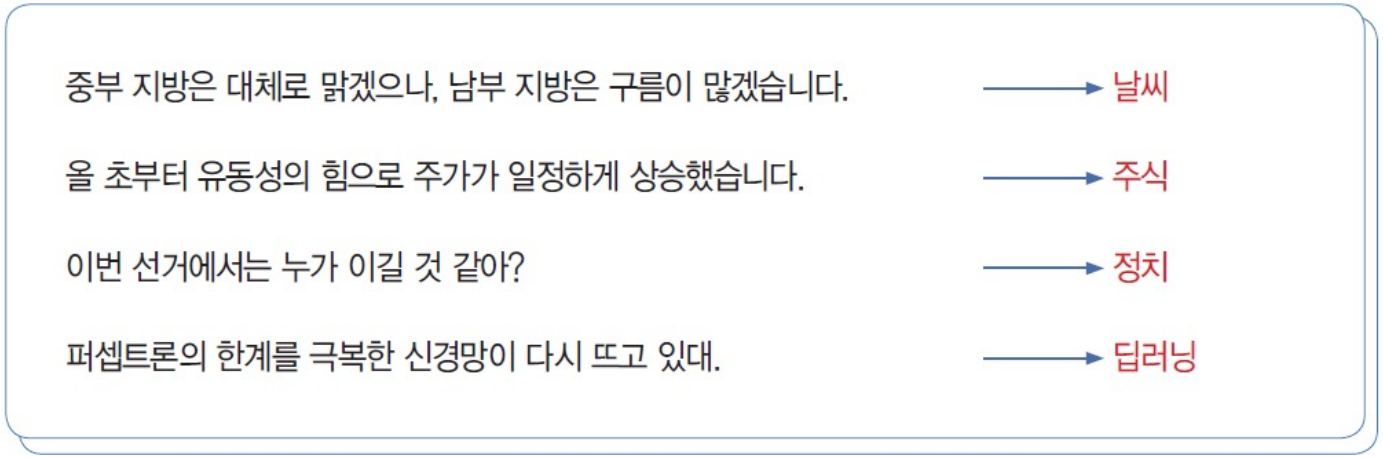
③ 다수 입력 다수 출력

예: 문장을 번역할 때 활용



LSTM으로 로이터 뉴스 카테고리 분류

- 입력된 문장의 의미를 파악하는 것은 모든 단어를 종합하여 하나의 카테고리로 분류하는 작업임



- 실습 : 로이터 뉴스를 읽고 어떤 의미를 지니는지 카테고리 분류

  - 로이터 뉴스 데이터 : 총 11,228개 뉴스 기사 -> 46개 카테고리

LSTM으로 로이터 뉴스 카테고리 분류

예제 소스: run_project/18_RNN1_Reuters.ipynb

```
import numpy
import tensorflow as tf
import matplotlib.pyplot as plt

# 로이터 뉴스 데이터셋 불러오기
from keras.datasets import reuters
from keras.models import Sequential
#from tensorflow.keras.models import Sequential
#from tensorflow.keras.layers import Dense,LSTM,Embedding
from keras.layers import Dense, LSTM, Embedding
from keras.preprocessing import sequence
from keras.utils import np_utils

# seed 값 설정
seed = 0
numpy.random.seed(seed)
tf.random.set_seed(3)

# 불러온 데이터를 학습셋, 테스트셋으로 나누기
(X_train, Y_train), (X_test, Y_test) = reuters.load_data(num_words=1000, test_split=0.2)

# 데이터 확인하기
category = numpy.max(Y_train) + 1
print(category, '카테고리')
print(len(X_train), '학습용 뉴스 기사')
print(len(X_test), '테스트용 뉴스 기사')
print(X_train[0])
```

첫 번째 뉴스 데이터는 단어들이 숫자 변환된 상태
빈도수 순서대로 단어에 번호를 붙인 상태(1은 빈도수가 1번째로 높은 단어라는 뜻)

```
46 카테고리
8982 학습용 뉴스 기사
2246 테스트용 뉴스 기사
[1, 2, 2, 8, 43, 10, 447, 5, 25, 207, 270, 5, 2, 111, 16, 369, 186, 90, 67, 7, 89, 5, 19, 102, 6, 19, 124, 15, 90, 6
7, 84, 22, 482, 26, 7, 48, 4, 49, 8, 864, 39, 209, 154, 6, 151, 6, 83, 11, 15, 22, 155, 11, 15, 7, 48, 9, 2, 2, 504,
6, 258, 6, 272, 11, 15, 22, 134, 44, 11, 15, 16, 8, 197, 2, 90, 67, 52, 29, 209, 30, 32, 132, 6, 109, 15, 17, 12]
```

LSTM으로 로이터 뉴스 카테고리 분류

- 모든 단어를 다 사용하는 것은 비효율적이므로 빈도가 높은 단어만 불러와 사용함

  maxlen=100 : 단어 수를 100개로 맞추라는 뜻

```
# 데이터 전처리
x_train = sequence.pad_sequences(X_train, maxlen=100)
x_test = sequence.pad_sequences(X_test, maxlen=100)
```

- y데이터에 원-핫 인코딩 처리

```
y_train = np_utils.to_categorical(Y_train)
y_test = np_utils.to_categorical(Y_test)
```

LSTM으로 로이터 뉴스 카테고리 분류

- 데이터 전처리 과정이 끝났으므로 딥러닝의 구조를 만들 차례임

```
# 모델의 설정
model = Sequential()
model.add(Embedding(1000, 100))
model.add(LSTM(100, activation='tanh'))
model.add(Dense(46, activation='softmax'))
```

- Embedding
  - 데이터 전처리 과정을 통해 입력된 값을 받아 다음 층이 알아들을 수 있는 형태로 변환하는 역할

    Embedding('불러온 단어의 총 개수', '기사당 단어 수'). 모델 설정 부분의 맨 처음에 있어야 함

- LSTM
  - RNN에서 기억 값에 대한 가중치를 제어하며, LSTM(기사당 단어 수, 기타 옵션)의 형식으로 적용됨
  - LSTM의 활성화 함수로는 tanh를 사용함

LSTM으로 로이터 뉴스 카테고리 분류

예제 소스: run_project/18_RNN1_Reuters.ipynb

```
# 모델의 설정
model = Sequential()
model.add(Embedding(1000, 300))  # 입력 벡터: 전체 단어의 인덱스(1000), 출력 벡터: 100
model.add(LSTM(100, activation='tanh'))      # LSTM(기사 당 단어 수=100, 옵션),  LSTM의 활성화함수로 tanh를 사용하면 성능이 좋게 나옴
model.add(Dense(46, activation='softmax'))

# 모델의 컴파일
model.compile(loss='categorical_crossentropy',
              optimizer='adam',
              metrics=['accuracy'])
```

```
# 자동 중단 설정
early_stopping_callback = EarlyStopping(monitor='val_loss', patience=3)
```

```
# 모델 저장 폴더 만들기
MODEL_DIR = './model/'
if not os.path.exists(MODEL_DIR):
    os.mkdir(MODEL_DIR)

modelpath = "./model/{epoch:02d}_{val_loss:.4f}.hdf5"

# 모델 업데이트 및 저장
checkpointer = ModelCheckpoint(filepath=modelpath, monitor='val_loss', verbose=1, save_best_only=True)
```

```
# 모델의 실행
history = model.fit(x_train_pad, y_train_onehot, batch_size=100, epochs=20, verbose=1, \
                    validation_data=(X_test_pad, Y_test_onehot), callbacks=[early_stopping_callback, checkpointer])
```

```
Epoch 00012: val_loss did not improve from 1.22444
90/90 [==============================] - 8s 86ms/step - loss: 0.8366 - accuracy: 0.7930 - val_loss: 1.2871 - val_accu
racy: 0.6874
Epoch 13/20
90/90 [==============================] - ETA: 0s - loss: 0.7922 - accuracy: 0.7989
Epoch 00013: val_loss did not improve from 1.22444
90/90 [==============================] - 8s 86ms/step - loss: 0.7922 - accuracy: 0.7989 - val_loss: 1.2266 - val_accu
racy: 0.7021
```

```
# 테스트 정확도 출력
print("\n Test Accuracy: %.4f" % (model.evaluate(X_test_pad, Y_test_onehot)[1]))
```

```
71/71 [==============================] - 1s 14ms/step - loss: 1.2266 - accuracy: 0.7021

 Test Accuracy: 0.7021
```