다섯째마당
딥러닝 활용하기

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

- 딥러닝을 이용한 자연어 처리

딥러닝을 이용한 자연어 처리

- **딥러닝을 이용한 자연어 처리**
  - 애플의 시리(siri), 구글의 어시스턴트(assistant), 아마존의 알렉사(alexa)나 네이버의 클로바(clova)까지 AI 비서라고 불리는 대화형 인공지능이 서로 경쟁하고 있음
  - 업무를 도와주고 삶의 질을 높여 주는 인공지능 비서 서비스를 누구나 사용하는 시대가 왔음
  - 스마트폰이나 스피커, 앱 같은 형태로 보급되는 인공지능 비서가 갖추어야 할 필수 능력은 사람의 언어를 이해하는 것
  - 문장을 듣고 무엇을 의미하는지 알아야 서비스를 제공해 줄 수 있기 때문임
  - 이 장에서는 이러한 능력을 만들어 주는 **자연어 처리**(Natural Language Processing, NLP)의 기본을 배울 것
  - 자연어란 우리가 평소에 말하는 음성이나 텍스트를 의미
  - 즉, 자연어 처리는 이러한 음성이나 텍스트를 컴퓨터가 인식하고 처리하는 것

딥러닝을 이용한 자연어 처리

- **딥러닝을 이용한 자연어 처리**
  - 컴퓨터를 이용해 인간의 말을 알아듣는 연구는 딥러닝이 나오기 이전부터 계속되어 왔음
  - 언어의 규칙은 컴퓨터의 규칙과 달리 쉽게 해결되지 않는 여러 문제를 안고 있었는데, 딥러닝이 등장하면서 자연어 처리 연구가 활발해지기 시작
  - 이는 대용량 데이터를 학습할 수 있는 딥러닝의 속성 때문임
  - 즉, 비교적 쉽게 얻을 수 있는 자연어 데이터를 지속적으로 입력해 끊임없이 학습하는 것이 가능해졌기 때문임
  - 텍스트 자료를 모았다고 해서 이를 딥러닝에 그대로 입력할 수 있는 것은 아님
  - 컴퓨터 알고리즘은 수치로 된 데이터만 이해할 뿐 텍스트는 이해할 수 없기 때문임
  - 텍스트를 정제하는 **전처리 과정**이 꼭 필요함
  - 여기서는 자연어 처리를 위해 텍스트를 전처리하는 과정부터 알아보자

1 텍스트의 토큰화

1 텍스트의 토큰화

- **텍스트의 토큰화**
  - 먼저 해야 할 일은 텍스트를 잘게 나누는 것
  - 입력할 텍스트가 준비되면 이를 단어별, 문장별, 형태소별로 나눌 수 있는데, 이렇게 작게 나누어진 하나의 단위를 **토큰**(token)이라고 함
  - 입력된 텍스트를 잘게 나누는 과정을 **토큰화**(tokenization)라고 함
  - 예를 들어 다음 문장이 주어졌다고 가정해 보자

```
'해보지 않으면 해낼 수 없다'
```

1 텍스트의 토큰화

- **텍스트의 토큰화**
  - 케라스가 제공하는 text 모듈의 text_to_word_sequence() 함수를 사용하면 문장을 단어 단위로 쉽게 나눌 수 있음
  - 해당 함수를 불러와 전처리할 텍스트를 지정한 후 다음과 같이 토큰화

```
from tensorflow.keras.preprocessing.text import text_to_word_sequence

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

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

1 텍스트의 토큰화

- **텍스트의 토큰화**
  - 결과는 다음과 같이 출력

실행 결과

```
원문:
 해보지 않으면 해낼 수 없다

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

1 텍스트의 토큰화

- **텍스트의 토큰화**
  - 이렇게 주어진 텍스트를 단어 단위로 쪼개고 나면 이를 이용해 여러 가지를 할 수 있음
  - 예를 들어 각 단어가 몇 번이나 중복해서 쓰였는지 알 수 있음
  - **단어의 빈도수를** 알면 텍스트에서 중요한 역할을 하는 단어를 파악할 수 있음
  - 텍스트를 단어 단위로 쪼개는 것은 가장 많이 쓰이는 전처리 과정

1 텍스트의 토큰화

- **텍스트의 토큰화**
  - Bag-of-Words라는 방법이 이러한 전처리를 일컫는 말인데, '단어의 가방(bag of words)'이라는 뜻 그대로, 같은 단어끼리 따로따로 가방에 담은 후 각 가방에 몇 개의 단어가 들어 있는지 세는 기법
  - 예를 들어 다음과 같은 세 개의 문장이 있다고 하자

> 먼저 텍스트의 각 단어를 나누어 토큰화합니다.
> 텍스트의 단어로 토큰화해야 딥러닝에서 인식됩니다.
> 토큰화한 결과는 딥러닝에서 사용할 수 있습니다.

1 텍스트의 토큰화

- **텍스트의 토큰화**
  - 이 세 문장에 등장하는 단어를 모두 세어 보고 출현 빈도가 가장 높은 것부터 나열해 보면, '토큰화'가 3회, '딥러닝에서'가 2회, '텍스트의'가 2회, 그리고 나머지 단어들이 1회씩
  - 가장 많이 사용된 단어인 '토큰화, 딥러닝, 텍스트'가 앞의 세 문장에서 중요한 역할을 하는 단어임을 짐작할 수 있음

1 텍스트의 토큰화

- **텍스트의 토큰화**
  - 케라스의 Tokenizer() 함수를 사용하면 단어의 빈도수를 쉽게 계산할 수 있음
  - 다음 예제는 위에 제시된 세 문장의 단어를 빈도수로 다시 정리하는 코드
  - 먼저 케라스에서 제공하는 텍스트 전처리 함수 중 Tokenizer() 함수를 불러옴

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

1 텍스트의 토큰화

- **텍스트의 토큰화**
  - 전처리하려는 세 개의 문장을 docs라는 배열에 지정

```
docs = ['먼저 텍스트의 각 단어를 나누어 토큰화합니다.',
        '텍스트의 단어로 토큰화해야 딥러닝에서 인식됩니다.',
        '토큰화한 결과는 딥러닝에서 사용할 수 있습니다.',
       ]
```

1 텍스트의 토큰화

- **텍스트의 토큰화**
  - 토큰화 함수인 Tokenizer()를 이용해 전처리하는 과정은 다음과 같음

```
token = Tokenizer()       # 토큰화 함수 지정
token.fit_on_texts(docs) # 토큰화 함수에 문장 적용

print("\n단어 카운트:\n", token.word_counts) # 단어의 빈도수를 계산한 결과 출력
```

1 텍스트의 토큰화

- **텍스트의 토큰화**
  - 마지막 줄에 있는 word_counts는 단어의 빈도수를 계산해 주는 함수
  - 이를 출력한 결과는 다음과 같음

실행 결과

```
단어 카운트:
 OrderedDict([('먼저', 1), ('텍스트의', 2), ('각', 1), ('단어를', 1), ('나누
어', 1), ('토큰화', 1), ('합니다', 1), ('단어로', 1), ('토큰화해야', 1), ('딥러
닝에서', 2), ('인식됩니다', 1), ('토큰화한', 1), ('결과는', 1), ('사용할', 1),
('수', 1), ('있습니다', 1)])
```

1 텍스트의 토큰화

- **텍스트의 토큰화**
  - '토큰화'가 3회, '텍스트의'와 '딥러닝에서'가 2회, 나머지가 1회씩 나오고 있음을 보여 줌
  - 순서를 기억하는 OrderedDict 클래스에 담겨 있는 형태로 출력되는 것을 볼 수 있음
  - Document_count( ) 함수를 이용하면 총 몇 개의 문장이 들어 있는지도 셀 수 있음

```
print("\n문장 카운트: ", token.document_count)
```

실행 결과

```
문장 카운트:  3
```

1 텍스트의 토큰화

- **텍스트의 토큰화**
  - word_docs() 함수를 통해 각 단어들이 몇 개의 문장에 나오는지 세어서 출력할 수도 있음
  - 출력되는 순서는 랜덤

```
print("\n각 단어가 몇 개의 문장에 포함되어 있는가:\n", token.word_docs)
```

1 텍스트의 토큰화

- 텍스트의 토큰화

실행 결과

```
각 단어가 몇 개의 문장에 포함되어 있는가:
 defaultdict(<class 'int'>, {'텍스트의': 2, '단어를': 1, '합니다': 1, '토
큰화': 1, '먼저': 1, '각': 1, '나누어': 1, '인식됩니다': 1, '딥러닝에서': 2,
'토큰화해야': 1, '단어로': 1, '수': 1, '사용할': 1, '결과는': 1, '있습니다': 1,
'토큰화한': 1})
```

1 텍스트의 토큰화

- **텍스트의 토큰화**
  - 각 단어에 매겨진 인덱스 값을 출력하려면 word_index( ) 함수를 사용

```
print("\n각 단어에 매겨진 인덱스 값:\n", token.word_index)
```

1 텍스트의 토큰화

- 텍스트의 토큰화

실행 결과

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

2 단어의 원-핫 인코딩

2 단어의 원-핫 인코딩

- **단어의 원-핫 인코딩**
  - 앞서 우리는 문장을 컴퓨터가 알아들을 수 있게 토큰화하고 단어의 빈도수를 확인해 보았음
  - 단순히 단어의 출현 빈도만 가지고는 해당 단어가 문장의 어디에서 왔는지, 각 단어의 순서는 어떠했는지 등에 관한 정보를 얻을 수 없음

2 단어의 원-핫 인코딩

- **단어의 원-핫 인코딩**
  - 단어가 문장의 다른 요소와 어떤 관계를 가지고 있는지 알아보는 방법이 필요함
  - 이러한 기법 중에서 가장 기본적인 방법인 **원-핫 인코딩**(one-hot encoding)을 알아보자
  - 앞서 '12장. 다중 분류 문제 해결하기'에서 배운 원-핫 인코딩과 같은 개념인데, 이것을 단어의 배열에 적용해 보는 것
  - 예를 들어 다음과 같은 문장이 있음

```
'오랫동안 꿈꾸는 이는 그 꿈을 닮아간다'
```

2 단어의 원-핫 인코딩

- **단어의 원-핫 인코딩**
  - 각 단어를 모두 0으로 바꾸어 주고 원하는 단어만 1로 바꾸어 주는 것이 원-핫 인코딩이었음
  - 이를 수행하기 위해 먼저 단어 수만큼 0으로 채워진 벡터 공간으로 바꾸면 다음과 같음

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

2 단어의 원-핫 인코딩

- **단어의 원-핫 인코딩**
  - 이제 각 단어가 배열 내에서 해당하는 위치를 1로 바꾸어서 벡터화할 수 있음

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

2 단어의 원-핫 인코딩

- **단어의 원-핫 인코딩**
  - 이러한 과정을 케라스로 실습해 보자
  - 먼저 토큰화 함수를 불러와 단어 단위로 토큰화하고 각 단어의 인덱스 값을 출력해 보자

```
text = "오랫동안 꿈꾸는 이는 그 꿈을 닮아간다"

token = Tokenizer()
token.fit_on_texts([text])
print(token.word_index)
```

2 단어의 원-핫 인코딩

- **단어의 원-핫 인코딩**
  - 결과는 다음과 같음

실행 결과

```
{'오랫동안': 1, '꿈꾸는': 2, '이는': 3, '그': 4, '꿈을': 5, '닮아간다': 6}
```

2 단어의 원-핫 인코딩

- **단어의 원-핫 인코딩**
  - 이제 각 단어를 원-핫 인코딩 방식으로 표현해 보자
  - 케라스에서 제공하는 Tokenizer의 texts_to_sequences( ) 함수를 사용해서 앞서 만들어진 토큰의 인덱스로만 채워진 새로운 배열을 만들어 줌

```
x = token.texts_to_sequences([text])
print(x)
```

실행 결과

```
[[1, 2, 3, 4, 5, 6]]
```

2 단어의 원-핫 인코딩

- **단어의 원-핫 인코딩**
  - 이제 1~6의 정수로 인덱스되어 있는 것을 0과 1로만 이루어진 배열로 바꾸어 주는 to_categorical() 함수를 사용해 원-핫 인코딩 과정을 진행
  - 배열 맨 앞에 0이 추가되므로 단어 수보다 1이 더 많게 인덱스 숫자를 잡아 주는 것에 유의하기 바람

```
from tensorflow.keras.utils import to_categorical

# 인덱스 수에 하나를 추가해서 원-핫 인코딩 배열 만들기
word_size = len(token.word_index) + 1
x = to_categorical(x, num_classes=word_size)

print(x)
```

2 단어의 원-핫 인코딩

- **단어의 원-핫 인코딩**
  - 결과는 다음과 같음
  - 예제 문장을 이루고 있는 단어들이 위에서부터 차례로 벡터화되었음

실행 결과

```
[[[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.]]]
```

오랫동안
꿈꾸는
이는
그
꿈을
닮아간다

3 단어 임베딩

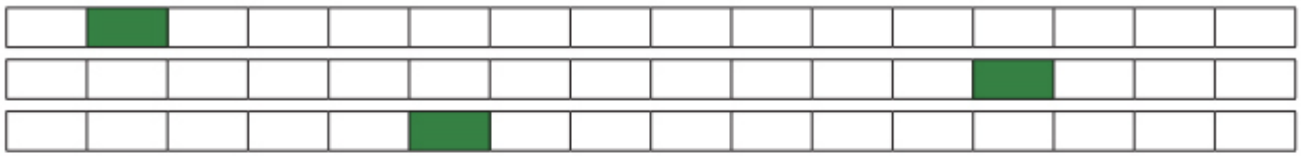
3 단어 임베딩

- **단어 임베딩**
  - 원-핫 인코딩과 함께 공부해야 할 것이 한 가지 더 있음
  - 원-핫 인코딩을 그대로 사용하면 벡터의 길이가 너무 길어진다는 단점이 있음
  - 예를 들어 1만 개의 단어 토큰으로 이루어진 말뭉치를 다룬다고 할 때, 이 데이터를 원-핫 인코딩으로 벡터화하면 9,999개의 0과 하나의 1로 이루어진 단어 벡터를 1만 개나 만들어야 함
  - 이러한 공간적 낭비를 해결하기 위해 등장한 것이 **단어 임베딩**(word embedding)이라는 방법

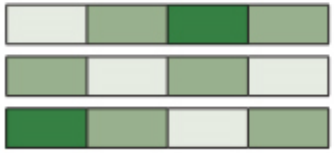
3 단어 임베딩

- **단어 임베딩**
  - 단어 임베딩은 주어진 배열을 정해진 길이로 압축시킴
  - 그림 17-1은 원-핫 인코딩을 사용해 만든 16차원 벡터가 단어 임베딩을 통해 4차원 벡터로 바뀐 예를 보기 쉽게 비교한 것

▼ **그림 17-1 | 원-핫 인코딩과 단어 임베딩**



▲ 원-핫 인코딩



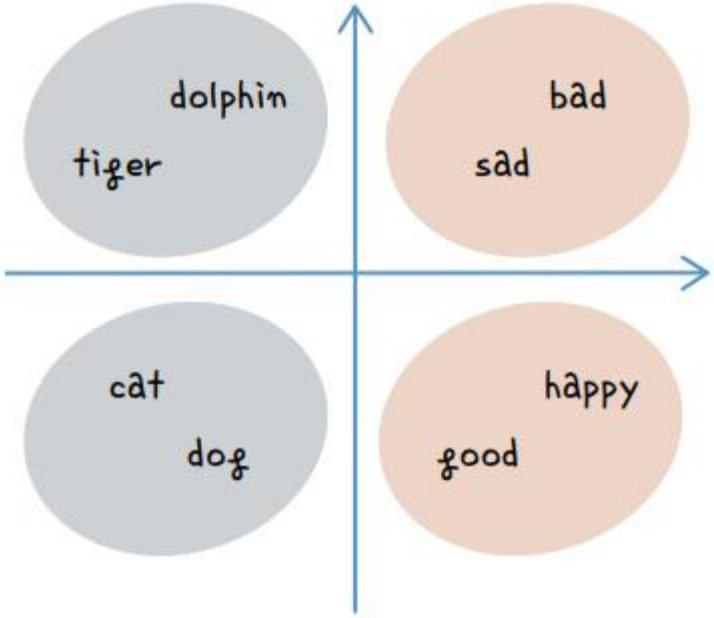
▲ 단어 임베딩

3 단어 임베딩

- **단어 임베딩**
  - 단어 임베딩으로 얻은 결과가 밀집된 정보를 가지고 있고 공간의 낭비가 적다는 것을 알 수 있음
  - 이러한 결과가 가능한 이유는 각 단어 간의 유사도를 계산했기 때문임
  - 예를 들어 happy라는 단어는 bad보다 good에 더 가깝고, cat이라는 단어는 good보다는 dog에 가깝다는 것을 고려해 각 배열을 새로운 수치로 바꾸어 주는 것(그림 17-2 참조)

3 단어 임베딩

▼ 그림 17-2 | 단어 간 유사도



3 단어 임베딩

- **단어 임베딩**
  - 그렇다면 이 단어 간 유사도는 어떻게 계산하는 것일까?
  - 여기서 앞서 배운 오차 역전파가 또다시 등장함
  - 적절한 크기로 배열을 바꾸어 주기 위해 최적의 유사도를 계산하는 학습 과정을 거치는 것
  - 이 과정은 케라스에서 제공하는 Embedding() 함수를 사용하면 간단히 해낼 수 있음
  - 예를 들어 다음과 같이 Embedding() 함수를 적용해 딥러닝 모델을 만들 수 있음

```
from tensorflow.keras.layers import Embedding

model = Sequential()
model.add(Embedding(16,4))
```

3 단어 임베딩

- **단어 임베딩**
  - Embedding() 함수는 입력과 출력의 크기를 나타내는 두 개의 매개변수가 있어야 함
  - 앞 예제에서 Embedding(16,4)는 입력될 총 단어 수는 16, 임베딩 후 출력되는 벡터 크기는 4로 하겠다는 의미
  - 여기에 단어를 매번 얼마나 입력할지 추가로 지정할 수 있음
  - Embedding(16, 4, input_length=2)라고 하면 총 입력되는 단어 수는 16개이지만 매번 두 개씩만 넣겠다는 의미
  - 단어 임베딩의 예는 다음 절에서 직접 실습하면서 확인할 것
  - 단어 임베딩을 포함하며 지금까지 배운 내용을 모두 적용해 텍스트 감정을 예측하는 딥러닝 모델을 만들어 보자

4 텍스트를 읽고 긍정, 부정 예측하기

4 텍스트를 읽고 긍정, 부정 예측하기

- **텍스트를 읽고 긍정, 부정 예측하기**
  - 실습해 볼 과제는 영화를 보고 남긴 리뷰를 딥러닝 모델로 학습해서 각 리뷰가 긍정적인지 부정적인지를 예측하는 것
  - 먼저 짧은 리뷰 열 개를 불러와 각각 긍정이면 1이라는 클래스를, 부정적이면 0이라는 클래스로 지정

```
# 텍스트 리뷰 자료를 지정합니다.
docs = ['너무 재밌네요','최고예요','참 잘 만든 영화예요','추천하고 싶은 영화입니다.','한 번 더 보고싶네요','글쎄요','별로예요','생각보다 지루하네요','연기가 어색해요','재미없어요']

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

4 텍스트를 읽고 긍정, 부정 예측하기

- **텍스트를 읽고 긍정, 부정 예측하기**
  - 그다음 앞서 배운 토큰화 과정을 진행
  - 케라스에서 제공하는 Tokenizer( ) 함수의 fit_on_texts를 이용해 각 단어를 하나의 토큰으로 변환

```
# 토큰화
token = Tokenizer()
token.fit_on_texts(docs)
print(token.word_index) # 토큰화된 결과를 출력해 확인합니다.
```

4 텍스트를 읽고 긍정, 부정 예측하기

- **텍스트를 읽고 긍정, 부정 예측하기**
  - 그럼 다음과 같이 출력

실행 결과

```
{'너무': 1, '재밌네요': 2, '최고예요': 3, '참': 4, '잘': 5, '만든': 6, '영화예요': 7, '추천하고': 8, '싶은': 9, '영화입니다': 10, '한번': 11, '더': 12, '보고싶네요': 13, '글쎄요': 14, '별로예요': 15, '생각보다': 16, '지루하네요': 17, '연기가': 18, '어색해요': 19, '재미없어요': 20}
```

4 텍스트를 읽고 긍정, 부정 예측하기

- **텍스트를 읽고 긍정, 부정 예측하기**
  - 이제 토큰에 지정된 인덱스로 새로운 배열을 생성

```
x = token.texts_to_sequences(docs)
print("\n리뷰 텍스트, 토큰화 결과:\n", x)
```

4 텍스트를 읽고 긍정, 부정 예측하기

- **텍스트를 읽고 긍정, 부정 예측하기**
  - 그럼 주어진 텍스트는 숫자로 이루어진 다음과 같은 배열로 재편됨

실행 결과

```
리뷰 텍스트, 토큰화 결과:
 [[1, 2], [3], [4, 5, 6, 7], [8, 9, 10], [11, 12, 13], [14], [15], [16,
17], [18, 19], [20]]
```

4 텍스트를 읽고 긍정, 부정 예측하기

- **텍스트를 읽고 긍정, 부정 예측하기**
  - 각 단어가 1부터 20까지의 숫자로 토큰화되었다는 것을 알 수 있음
  - 입력된 리뷰 데이터의 토큰 수가 각각 다름
  - 예를 들어 '최고예요'는 하나의 토큰([3])이지만 '참 잘 만든 영화예요'는 네 개의 토큰([4, 5, 6, 7])을 가지고 있음
  - 딥러닝 모델에 입력하려면 학습 데이터의 길이가 동일해야 함
  - 토큰의 수를 똑같이 맞추어 주어야 함

4 텍스트를 읽고 긍정, 부정 예측하기

- **텍스트를 읽고 긍정, 부정 예측하기**
  - 이처럼 길이를 똑같이 맞추어 주는 작업을 **패딩**(padding) 과정이라고 함
  - 패딩은 자연어 처리뿐만 아니라 19장에서 소개할 GAN에서도 중요한 역할을 하니 잘 기억하기 바람
  - 패딩 작업을 위해 케라스는 pad_sequences( ) 함수를 제공
  - pad_sequences() 함수를 사용하면 원하는 길이보다 짧은 부분은 숫자 0을 넣어서 채워 주고, 긴 데이터는 잘라서 같은 길이로 맞춤
  - 앞에서 생성한 x 배열을 네 개의 길이로 맞추기 위해 다음과 같이 실행

```
padded_x = pad_sequences(x, 4) # 서로 다른 길이의 데이터를 4로 맞추기
print("\n패딩 결과:\n", padded_x)
```

4 텍스트를 읽고 긍정, 부정 예측하기

- **텍스트를 읽고 긍정, 부정 예측하기**
  - 다음과 같이 배열의 길이가 맞추어짐

실행 결과

```
패딩 결과:
 [[ 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]]
```

4 텍스트를 읽고 긍정, 부정 예측하기

- **텍스트를 읽고 긍정, 부정 예측하기**
  - 이제 단어 임베딩을 포함해 딥러닝 모델을 만들고 결과를 출력해 보자
  - 임베딩 함수에 필요한 세 가지 파라미터는 '입력, 출력, 단어 수'
  - 총 몇 개의 단어 집합에서(입력), 몇 개의 임베딩 결과를 사용할 것인지(출력), 그리고 매번 입력될 단어 수는 몇 개로 할지(단어 수)를 정해야 하는 것

4 텍스트를 읽고 긍정, 부정 예측하기

- **텍스트를 읽고 긍정, 부정 예측하기**
  - 먼저 총 몇 개의 인덱스가 '입력'되어야 하는지 정함
  - word_size라는 변수를 만든 후 길이를 세는 len() 함수를 이용해 word_index 값을 앞서 만든 변수에 대입
  - 이때 전체 단어의 맨 앞에 0이 먼저 나와야 하므로 총 단어 수에 1을 더하는 것을 잊지 마시기 바람

```
word_size = len(token.word_index) + 1
```

4 텍스트를 읽고 긍정, 부정 예측하기

- **텍스트를 읽고 긍정, 부정 예측하기**
  - 이제 몇 개의 임베딩 결과를 사용할 것인지, 즉 '출력'을 정할 차례
  - 이번 예제에서는 word_size만큼 입력 값을 이용해 여덟 개의 임베딩 결과를 만들겠음
  - 여기서 8이라는 숫자는 임의로 정한 것
  - 데이터에 따라 적절한 값으로 바꿀 수 있음
  - 이때 만들어진 여덟 개의 임베딩 결과는 우리 눈에 보이지 않음
  - 내부에서 계산해 딥러닝의 층으로 활용
  - 끝으로 매번 입력될 '단어 수'를 정함
  - 패딩 과정을 거쳐 네 개의 길이로 맞추어 주었으므로 네 개의 단어가 들어가게 설정하면 임베딩 과정은 다음 한 줄로 표현

```
Embedding(word_size, 8, input_length=4)
```

4 텍스트를 읽고 긍정, 부정 예측하기

- **텍스트를 읽고 긍정, 부정 예측하기**
  - 이를 이용해 모델을 만들면 다음과 같음

```
# 단어 임베딩을 포함해 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]))
```

4 텍스트를 읽고 긍정, 부정 예측하기

- **텍스트를 읽고 긍정, 부정 예측하기**
  - 최적화 함수로 adam()을 사용하고 오차 함수로는 binary_crossentropy()를 사용
  - 20번 반복하고 나서 정확도를 계산해 출력하게 했음

4 텍스트를 읽고 긍정, 부정 예측하기

- **텍스트를 읽고 긍정, 부정 예측하기**
  - 지금까지 모든 과정을 한눈에 보면 다음과 같음

실습 | 영화 리뷰가 긍정적인지 부정적인지를 예측하기

```
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
from tensorflow.keras.utils import to_categorical

from numpy import array
```

4 텍스트를 읽고 긍정, 부정 예측하기

- 텍스트를 읽고 긍정, 부정 예측하기

```
# 텍스트 리뷰 자료를 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)
```

4 텍스트를 읽고 긍정, 부정 예측하기

- 텍스트를 읽고 긍정, 부정 예측하기

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

# 임베딩에 입력될 단어의 수를 지정합니다.
word_size = len(token.word_index) + 1
# 단어 임베딩을 포함해 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.summary()
```

4 텍스트를 읽고 긍정, 부정 예측하기

- 텍스트를 읽고 긍정, 부정 예측하기

```
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]))
```

4 텍스트를 읽고 긍정, 부정 예측하기

- 텍스트를 읽고 긍정, 부정 예측하기

실행 결과

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding (Embedding)        (None, 4, 8)              168
_________________________________________________________________
flatten (Flatten)            (None, 32)                0
_________________________________________________________________
dense (Dense)                (None, 1)                 33
=================================================================
Total params: 201
Trainable params: 201
Non-trainable params: 0
```

4 텍스트를 읽고 긍정, 부정 예측하기

- 텍스트를 읽고 긍정, 부정 예측하기

```
_________________________________________________________________
Epoch 1/20
1/1 [==============================] - 0s 325ms/step - loss: 0.6903 -
accuracy: 0.5000
... (중략) ...
Epoch 20/20

1/1 [==============================] - 0s 997us/step - loss: 0.6596 -
accuracy: 1.0000
1/1 [==============================] - 0s 92ms/step - loss: 0.6579 -
accuracy: 1.0000
Accuracy: 1.0000
```

- 긍정적인 리뷰와 부정적인 리뷰의 학습이 진행되는 것을 확인할 수 있음