

딥러닝을 이용한 자연어 처리

1. 텍스트의 토큰화

```
In [3]: 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

# 케라스의 텍스트 전처리와 관련한 함수 중 text_to_word_sequence 함수를 불러옵니다.
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)
```

원문:

해보지 않으면 해낼 수 없다

토큰화:

['해보지', '않으면', '해낼', '수', '없다']

```
In [4]: # 단어 빈도수 세기

# 전처리하려는 세 개의 문장을 정합니다.
docs = ['먼저 텍스트의 각 단어를 나누어 토큰화합니다.',
        '텍스트의 단어로 토큰화해야 딥러닝에서 인식됩니다.',
        '토큰화한 결과는 딥러닝에서 사용할 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.
# Tokenizer()의 word_counts 함수는 순서를 기억하는 OrderedDict 클래스를 사용합니다.
print("\n단어 카운트:\n", token.word_counts)

# 출력되는 순서는 랜덤입니다.
print("\n문장 카운트: ", token.document_count)
print("\n각 단어가 몇 개의 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, '토큰화합니다': 1, '단어로': 1, '토큰화해야': 1, '딥러닝에서': 2, '인식됩니다': 1, '토큰화한': 1, '결과는': 1, '사용할': 1, '수': 1, '있습니다': 1})
```

문장 카운트: 3

각 단어가 몇 개의 문장에 포함되어 있는가:

```
defaultdict(<class 'int'>, {'각': 1, '먼저': 1, '나누어': 1, '토큰화합니다': 1, '단어를': 1, '텍스트의': 2, '단어로': 1, '인식됩니다': 1, '토큰화해야': 1, '딥러닝에서': 2, '토큰화한': 1, '수': 1, '있습니다': 1, '사용할': 1, '결과는': 1})
```

각 단어에 매겨진 인덱스 값:

```
{'텍스트의': 1, '딥러닝에서': 2, '먼저': 3, '각': 4, '단어를': 5, '나누어': 6, '토큰화합니다': 7, '단어로': 8, '토큰화해야': 9, '인식됩니다': 10, '토큰화한': 11, '결과는': 12, '사용할': 13, '수': 14, '있습니다': 15}
```

2. 단어의 원-핫 인코딩

```
In [6]: text="오랫동안 꿈꾸는 이는 그 꿈을 닮아간다"
token = Tokenizer()
token.fit_on_texts([text])
print(token.word_index)
```

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

```
In [7]: x=token.texts_to_sequences([text])
print(x)
```

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

```
In [8]: # 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)
```

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

3. 텍스트를 읽고 긍정, 부정 예측하기

```
In [10]: # 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)
```

```
{'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}
```

```
In [11]: x = token.texts_to_sequences(docs)
print("\n리뷰 텍스트, 토큰화 결과:\n", x)
```

리뷰 텍스트, 토큰화 결과:

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

```
In [12]: # 패딩, 서로 다른 길이의 데이터를 4로 맞추어 줍니다.
padded_x = pad_sequences(x, 4)
print("\n패딩 결과:\n", padded_x)
```

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]]
```

```
In [13]: # 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))
model.add(Flatten())
model.add(Dense(1, activation='sigmoid'))
model.build(input_shape=(None,4))
model.summary()
```

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 (804.00 B)


Trainable params: 201 (804.00 B)


Non-trainable params: 0 (0.00 B)


```
In [14]: 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 1/20
1/1  0s 456ms/step - accuracy: 0.7000 - loss: 0.6829


Epoch 2/20
1/1  0s 34ms/step - accuracy: 0.7000 - loss: 0.6807


Epoch 3/20
1/1  0s 29ms/step - accuracy: 0.8000 - loss: 0.6784


Epoch 4/20
1/1  0s 29ms/step - accuracy: 0.8000 - loss: 0.6762


Epoch 5/20
1/1  0s 30ms/step - accuracy: 0.8000 - loss: 0.6740


Epoch 6/20
1/1  0s 29ms/step - accuracy: 0.8000 - loss: 0.6718


Epoch 7/20
1/1  0s 38ms/step - accuracy: 0.8000 - loss: 0.6696


Epoch 8/20
1/1  0s 30ms/step - accuracy: 0.8000 - loss: 0.6674


Epoch 9/20
1/1  0s 95ms/step - accuracy: 0.8000 - loss: 0.6652


Epoch 10/20
1/1  0s 29ms/step - accuracy: 0.8000 - loss: 0.6629


Epoch 11/20
1/1  0s 36ms/step - accuracy: 0.8000 - loss: 0.6607


Epoch 12/20
1/1  0s 29ms/step - accuracy: 0.9000 - loss: 0.6584


Epoch 13/20
1/1  0s 30ms/step - accuracy: 0.8000 - loss: 0.6562


Epoch 14/20
1/1  0s 29ms/step - accuracy: 0.8000 - loss: 0.6539



Epoch 15/20
1/1  0s 64ms/step - accuracy: 0.8000 - loss: 0.6516

Epoch 16/20
1/1  0s 59ms/step - accuracy: 0.9000 - loss: 0.6494

Epoch 17/20
1/1  0s 28ms/step - accuracy: 0.9000 - loss: 0.6471

Epoch 18/20
1/1  0s 28ms/step - accuracy: 0.9000 - loss: 0.6448

Epoch 19/20
1/1  0s 30ms/step - accuracy: 0.9000 - loss: 0.6425

Epoch 20/20
1/1  0s 30ms/step - accuracy: 0.9000 - loss: 0.6402
1/1  0s 97ms/step - accuracy: 0.9000 - loss: 0.6379

Accuracy: 0.9000

In []: