11장

https://velog.io/@conel77/케라스-창시자에게-배우는-딥러닝11장#성능-향상시키기--단어-임베딩-이해하기-

단어 수준 토큰화: 단어를 부분 단어로 나누기 - staring → star + ing

N - 그램 토큰화: 토큰이 N개의 연속된 단어이다. ex “this is” “is a” “a sample”

n 그램 문제 예시

텍스트: “I love AI study” 를 1-그램, 2-그램으로 분석하시오
1-grams: [”I”, ”love”, ”AI”, ”study”]
2-grams: [”I love”, “love AI”, “AI study”]

N-grams: 단어의 순서를 고려하여 텍스트를 연속된 N개의 묶음으로표현
Bow: 단어의 순서를 무시하고 단어의 빈도 수로 표현하는 기법, 단어 빈도를 벡터로 표현
ex [I love NLP and I love AI] → [2,2,1,1,1]

시퀀스 모델 - 단어의 순서를 고려한다. + 단어 수준 토큰화

Bow 모델 - 단어의 순서를 무시한다. + N 그램 토큰화

---

page 24

```
from tensorflow.keras.layers import TextVectorization
text_verctorization = TextVectorization(
        output_mode = "int"
)
```

output_mode = “int”의 의미 - 인코딩된 단어 시퀀스를 (사전의) 정수 인덱스로 반환하도록 설정한다.
* 벡터화 된 텍스트는 정수 시퀀스로 표현한다.

사전 10000개에 없는 드문 단어는 예외 어휘로서 OOV 인덱스인 1을 사용한다. 이 1을 디코딩 할때는 [UNK]로 디코딩한다. 참고로 0은 이미 사용되는 토큰이라 1을 사용한다.

---

page 30

훈련 텍스트의 20%를 validation으로 덜어낸다.

```
import os, pathlib, shutil, random

base_dir = pathlib.Path("aclImdb")
val_dir = base_dir / "val"
train_dir = base_dir / "train"
for category in ("neg", "pos"):
    os.makedirs(val_dir / category)
    files = os.listdir(train_dir / category)
    random.Random(1337).shuffle(files)
    num_val_samples = int(0.2 * len(files))
    val_files = files[-num_val_samples:]
    for fname in val_files:
        shutil.move(train_dir / category / fname,
                    val_dir / category / fname)
```

코드를 여러 번 실행해도 동일한 검증 세트가 만들어지도록 랜덤 시드를 지정하여 훈련 파일 목록을 섞습니다.

훈련 파일 중 20%를 검증 세트로 덜어 냅니다.

파일을 aclImdb/val/neg와 aclImdb/val/pos로 옮깁니다.

이 과정(**text_dataset_from_directory**)에서 **클래스에 따라 레이블이 positive인지 negative인지 자동으로 생성**된다. (두 번 말함)

- text_dataset_from_directory - 디렉터리 구조로 구성된 텍스트 파일에서 데이터 세트를 만드는 데 사용
- **Create a dataset** - 훈련, 검증, 테스트를 위한 3개의 Dataset 객체 생성

```
from tensorflow import keras
batch_size = 32

train_ds = keras.utils.text_dataset_from_directory(
    "aclImdb/train", batch_size=batch_size
)
val_ds = keras.utils.text_dataset_from_directory(
    "aclImdb/val", batch_size=batch_size
)
test_ds = keras.utils.text_dataset_from_directory(
    "aclImdb/test", batch_size=batch_size
)
```

이 라인을 실행하면 "Found 20000 files belonging to 2 classes"가 출력될 것입니다. "Found 70000 files belonging to 3 classes"처럼 출력된다면 aclImdb/train/unsup 디렉터리를 삭제하고 다시 실행하세요.

tf.Tesonr(b"This string contains the movie review"에서 b는 그냥 byte 자료형을 나타냄

batch_size 가 32여서 inputs.shape(타입 string)이 (32,)임 targets.shape은 클래스 레이블(긍정 0, 부정 1)을 나타내므로 타입이 int32이다.

코드 11-2 첫 번째 배치의 크기와 dtype 출력하기

```
>>> for inputs, targets in train_ds:
>>>     print("inputs.shape:", inputs.shape)
>>>     print("inputs.dtype:", inputs.dtype)
>>>     print("targets.shape:", targets.shape)
>>>     print("targets.dtype:", targets.dtype)
>>>     print("inputs[0]:", inputs[0])
>>>     print("targets[0]:", targets[0])
>>>     break
inputs.shape: (32,)
inputs.dtype: <dtype: "string">
targets.shape: (32,)
targets.dtype: <dtype: "int32">
inputs[0]: tf.Tensor(b"This string contains the movie review.", shape=(), dtype=string)
targets[0]: tf.Tensor(1, shape=(), dtype=int32)
```

32개의 텍스트 샘플 (string)

해당 텍스트 샘플의 클래스 레이블을 나타내는 정수

b - 바이트 문자열

TextVectorization 층으로 전처리

page 33

output_mode 가 “int”에서 “multi_hot”으로 변경됨 [0,1,1,0 ...0] 총 20000개의 인덱스 번호에 0 or 1을 표현함. (이진 인코딩)

lambda x, y : x 를 사용해서 레이블은 없애버리고 원시 텍스트 입력만 반환하도록 한다. (걍 암기)

x는 reivew를 의미한다.

text_vectorization.adapt - adpat() 메서드로 데이터셋을 어휘 사전에 인덱싱한다.

text-Vertorization(x) - 멀티 핫 인코딩으로 변환
(text_Vectorization(x), y) - 텍스트 데이터가 멀티 핫 인코딩으로 변환되지만 레이블은동일하게유지되는튜플을반환

코드 11-3 TextVectorization 층으로 데이터 전처리하기

```
text_vectorization = TextVectorization(
    max_tokens=20000,
    output_mode="multi_hot",
)
text_only_train_ds = train_ds.map(lambda x, y: x)
text_vectorization.adapt(text_only_train_ds)
binary_1gram_train_ds = train_ds.map(
    lambda x, y: (text_vectorization(x), y),
    num_parallel_calls=4)
```

가장 많이 등장하는 2만 개 단어로 어휘 사전을 제한합니다. 그렇지 않으면 훈련 데이터에 있는 모든 단어를 인덱싱하게 됩니다. 아마도 수만 개의 단어가 한 번 또는 두 번만 등장하면 유용하지 않을 것입니다. 일반적으로 텍스트 분류에서 2만 개는 적절한 어휘 사전 크기입니다.

멀티-핫 이진 벡터로 출력 토큰을 인코딩합니다.

(레이블 없이) 원시 텍스트 입력만 반환하는 데이터셋을 준비합니다.

text, label 쌍

x (text) 반환

adapt() 메서드로 이 데이터셋의 어휘 사전을 인덱싱합니다.

효율적인 텍스트-숫자 변환을 준비

text_Vectorization(x) - 멀티 핫 인코딩으로 변환
(text_Vectorization(x), y) - 텍스트 데이터가 변환되지만
레이블은 동일하게 유지되는 튜플을 반환

사전 수를 20000개로 제한하는 이유 - 제한하지 않으면 훈련 데이터에 있는 모든 단어를 인덱싱하게된다. 한 번 or 두 번만 등장하는 단어는 굳이 인덱싱할 필요가 때문에 자주 등장하는 2만개로 제한을 둔다.

num_parallel_calss = 4는 다중 CPU 코어를 사용하기 위해 사용한다.

page 35

2만개의 리뷰가 32개 배치사이즈로 저장되어 inputs.shape: (32, 20000)임

target은 0아니면 1이므로 targets.shape(32,)

코드 11-4 이진 유니그램 데이터셋의 출력 확인하기

```
>>> for inputs, targets in binary_1gram_train_ds:
>>>     print("inputs.shape:", inputs.shape)
>>>     print("inputs.dtype:", inputs.dtype)
>>>     print("targets.shape:", targets.shape)
>>>     print("targets.dtype:", targets.dtype)
>>>     print("inputs[0]:", inputs[0])
>>>     print("targets[0]:", targets[0])
>>>     break
inputs.shape: (32, 20000)
inputs.dtype: <dtype: "float32">
targets.shape: (32,)
targets.dtype: <dtype: "int32">
inputs[0]: tf.Tensor([1. 1. 1. ... 0. 0. 0.], shape=(20000,), dtype=float32)
targets[0]: tf.Tensor(1, shape=(), dtype=int32)
```

입력은 20,000차원 벡터의 배치입니다.

이런 벡터는 전부 0과 1로 구성됩니다.

cahce 사용

page 37

```
model.fit(binary_1gram_train_ds.cache(),
          validation_data=binary_1gram_val_ds.cache(),
          epochs=10,
          callbacks=callbacks)
```

리에 캐싱합니다. 이렇게 하면 첫 번째 에포크에서 한 번만 전처리하고 이후 에포크에서는 전처리된 텍스트를 재사용합니다. 메모리에 들어갈 만큼 작은 데이터일 때 사용할 수 있습니다.

cahce() 메서드를 사용하면 첫 번째 에포크에서 한 번만 전처리하고 이후 에포크에서는 전처리된 텍스트를 **재사용**한다.

정확도는 89.2% - 단어를 집합으로 처리

여기까지는 1그램 방식 (유니그램)

이후 부터는 2그램 방식 (바이그램)

page 40에서 ngrams=2로 되어있는 모습

natural language가 0번째 인덱스
language processing이 1번째 인덱스로 들어가 두 덩이를 하나의 단어로 인식해 끊어서 인덱스에 저장된다.

Counter는 하나의 단어로 인식한 두덩이의 빈도수를 세는 것. natural language는 두 번 등장함

결과 - Bow 방식(단어를 집합으로 처리)

테스트 정확도 90.4%
이는 두덩이를 하나로 인식해 문맥을 생성한 것. → 국부적인 순서가 중요하다는 것을 정확도 상승이 이야기해준다.

코드 11-7 바이그램을 반환하는 TextVectorization 층 만들기

```
text_vectorization = TextVectorization(
    ngrams=2,
    max_tokens=20000,
    output_mode="multi_hot",
)
```

text = "Natural language processing is fascinating and natural language understanding is crucial."

{ 'natural language': 0,
'language processing': 1,
'processing is': 2, 'is fascinating': 3, 'fascinating and': 4, 'and natural': 5, 'language understanding': 6,
'understanding is': 7, 'is crucial': 8 }

텍스트의 인덱스 표현: [0, 1, 2, 3, 4, 5, 0, 6, 7, 8]

Counter({'natural language': 2, 'language processing': 1, 'processing is': 1, 'is fascinating': 1, 'fascinating and': 1, 'and natural': 1, 'language understanding': 1, 'understanding is': 1, 'is crucial': 1})

예시
text = “Natural language processing is fascinating and natural language understanding is crucial

→ 텍스트의 인덱스 표현: [0, 1, 2, 3, 4, 5, 0, 6, 7, 8]

→ Counter({‘natural language’: 2, ….. 나머지는 1})

---

page 44

자주 등장하는데 의미 없는 단어 ex관사, 전치사, 연결사 같은 단어의 중요도를 떨어트리기 위해 TF-IDF방식을 사용한다.

빈번도를 계산하고, 빈번도의 역수의 합에 로그를 취한 값으로 나눈다. (개념만 참고해서 알아둘것)

ex 빈번도가 10

빈번도 역수 1/10

여기서 로그를 안취하고 역수의합을 나누면 의미가 없음.

10 / (1/10 * 10) = 10

그래서 10 / log(1/10 * 10) 을 사용해 쓸데없이 많이 증가할 수록 중요도가 떨어지게된다.

output_mode 매개변수를 "tf_idf"로 바꾸기만 하면 사용할 수 있음

코드 11-10 TF-IDF 가중치가 적용된 출력을 반환하는 TextVectorization 층

```
text_vectorization = TextVectorization(
    ngrams=2,
    max_tokens=20000,
    output_mode="tf_idf",
)
```

코딩할 때는 page 48처럼 output_mode="tf_idf" 사용

사용 결과 정확도는 89.8%로 그닥 도움은 안되어 보이지만 데이터 양이 적은 이유도 있고 대부분 이진 인코딩보다 TF-IDF를 사용하는게 1퍼센트 정도 성능이 높아짐

5월 29일 11 - b 강의

바이그램 - '수동'으로 만든 순서 기반의 특성 사용

원시 단어 시퀀스를 (바이그램 사용 전) 모델에 전달해 스스로 특성을 학습하도록 하는것이 시퀀스 모델이다.

원핫인코딩으로 단어를 표현해서 모델을 돌리면 성능이 그닥 좋지 않은 이유는

원핫 인코딩은 단어를 [0,0,0,1,0] 이런식으로 1이 하나만 들어가서 표현하는데 1의 위치는 단어가 다르면 절대 같을 수가 없다. 따라서 원 핫 인코딩으로 표현한 단어끼리는 무조건 수직이다.

벡터에서 수직은 의미상 연관이 없음을 의미하게 되는데 실제로 어떤 단어들은 서로 비슷한 의미를 가지고 있다. 원핫인코딩은 이런 연관성을 무시하는 기법이므로 성능이 떨어질 수 밖에없다.

즉, 원-핫 인코딩이나 해싱으로 얻은 단어 표현은 희소하고, 고차원이며 하드코딩되어있다.

단어 임베딩은 다른 의미를 가지는 단어는 서로 멀리 떨어지게하고 관련 있는 단어는 가까이 놓여있도록 하는 단어 벡터의 표현입니다. 비교적 저차원이고 데이터로부터 학습합니다.

코드 11-16 밑바닥부터 훈련하는 Embedding 층을 사용한 모델

```
inputs = keras.Input(shape=(None,), dtype="int64")
embedded = layers.Embedding(input_dim=max_tokens, output_dim=256)(inputs)
x = layers.Bidirectional(layers.LSTM(32))(embedded)
x = layers.Dropout(0.5)(x)
outputs = layers.Dense(1, activation="sigmoid")(x)
model = keras.Model(inputs, outputs)
model.compile(optimizer="rmsprop",
              loss="binary_crossentropy",
              metrics=["accuracy"])
model.summary()
```

```
callbacks = [
    keras.callbacks.ModelCheckpoint("embeddings_bidir_lstm.keras",
                                    save_best_only=True)
]
```

단어를 임베딩하는 방법은 학습

학습한 임베딩 방법을 모델에 적용(Embedding 층을 만들어 단어 임베딩 학습)
또는 사전에 이미 만들어둔 임베딩 모델 사용

즉, 임베딩층은 정수 인덱스를 밀집 벡터로 매핑하는 딕셔너리 역할이다.

```
embedding_layer = layers.Embedding(input_dim=max_tokens, output_dim=256)
#임베딩 층은 적어도 2개의 매개변수가 필요하다. 가능한 토큰의 개수와 임베딩 차원이다.
```

입력 형태는 (32, 600)
임베딩 벡터 출력의 형태는 (32, 600, 256) 이다.

```
inputs = keras.Input(shape=(None,), dtype="int64")
embedded = layers.Embedding(input_dim=max_tokens, output_dim=256)(inputs)
x = layers.Bidirectional(layers.LSTM(32))(embedded)
x = layers.Dropout(0.5)(x)
outputs = layers.Dense(1, activation="sigmoid")(x)
model = keras.Model(inputs, outputs)
model.compile(optimizer="rmsprop",
              loss="binary_crossentropy",
              metrics=["accuracy"])
model.summary()

callbacks = [
    keras.callbacks.ModelCheckpoint("embeddings_bidir_lstm.keras",
                                    save_best_only=True)
]
model.fit(int_train_ds, validation_data=int_val_ds, epochs=10, callbacks=callbacks)
model = keras.models.load_model("embeddings_bidir_lstm.keras")
print(f"테스트 정확도: {model.evaluate(int_test_ds)[1]:.3f}")
```

테스트 정확도는 87%가 나왔고, LSTM이 20000차원이 아니라 256차원의 벡터를 처리하기 때문에 이 모델은 원-핫 모델보다 훨씬 빠르고 정확도는 비슷하다. 정확도가 제대로 개선이 안된 이유는 모델이 약간 적은 데이터를 사용했기 때문이다. 바이그램 모델은 전체 리뷰를 처리하지만 ***이 시퀀스 모델은 600개의 단어 이후 시퀀스를 잘라버린다.. (정보의 유실을 지적)***

- 기본적으로 이 옵션은 활성화되어 있지 않음
- 이를 활성화하려면 Embedding 층에 mask_zero=True를 지정
- Embedding 층의 compute_mask() 메서드로 입력에 대한 마스킹 추출
- 마스킹을 사용하여 RNN 층은 마스킹된 스텝을 건너뜀

```
>>> embedding_layer = Embedding(input_dim=10, output_dim=256, mask_zero=True)
>>>  some_input = [
...  [4, 3, 2, 1, 0, 0, 0],
...  [5, 4, 3, 2, 1, 0, 0],
...  [2, 1, 0, 0, 0, 0, 0]]
>>> mask = embedding_layer.compute_mask(some_input)

<tf.Tensor: shape=(3, 7), dtype=bool, numpy=
array([[ True,  True,  True,  True, False, False, False],
       [ True,  True,  True,  True,  True, False, False],
       [ True,  True, False, False, False, False, False]])>
```

TextVectorization 층에 output_sequence_length = max_length로 인해 토큰 길이 만큼 잘린다.

mask_zero는 600개의 토큰보다 긴 문장은 600개의 토큰 길이(max_length)로 잘리고 600개의 토큰보다 짧은 문장은 끝에 0으로 채운다.

이 옵션은 mask_zero = True 를 사용하면 사용할 수 있고 디폴트는 False이다.
0이 아닌값을 True, 0을 False로 표시하는 입력에 대한 마스킹을 추출한다. 이 마스킹을 사용해서 모델이 True인 값만 학습하도록 해 모델에 성능을 높인다. (zero padding)