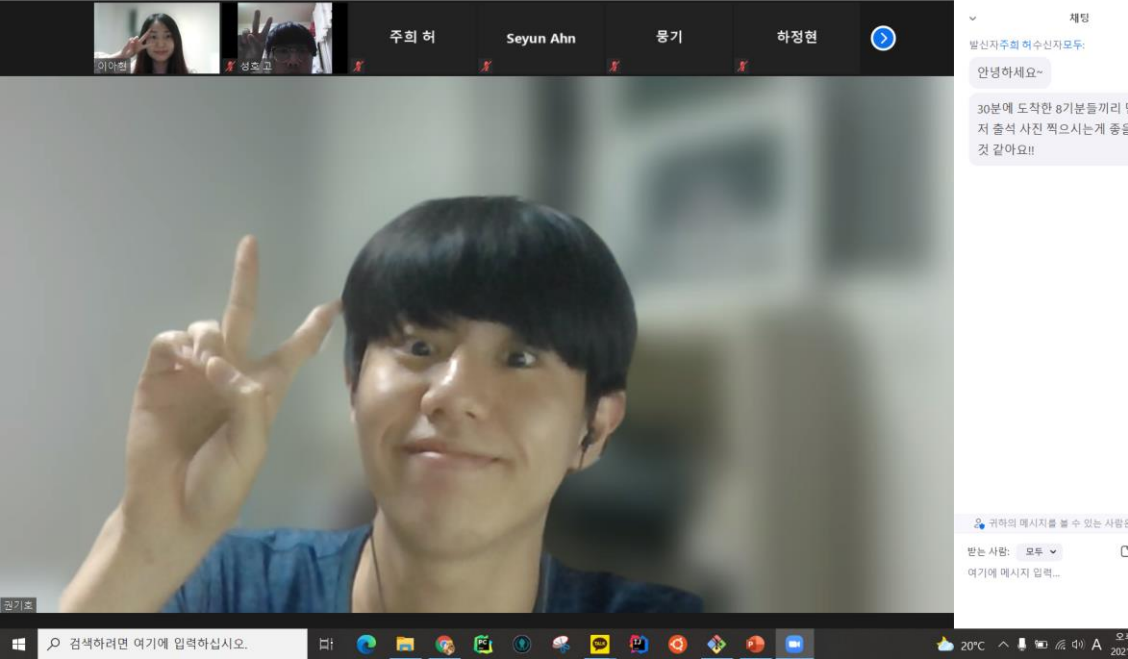
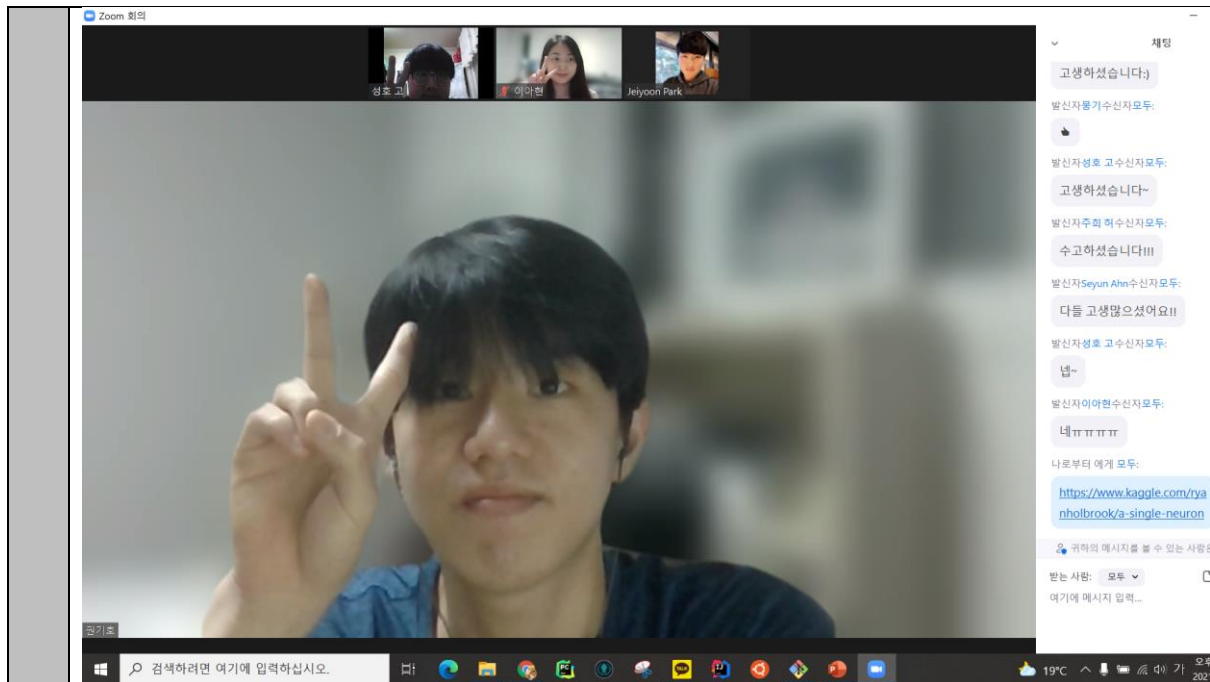


TAVE 서기

서기 내용			
서기 일자	21.10.08	서기	고성호
주제	텐서플로에서 데이터 적재와 전처리		
시간	20:30~22:30	장소	온라인
스터디 인원	<p>고성호, 권기호, 이아현: 시작</p> 		
	<p>고성호, 권기호, 이아현 : 종료</p>		



내용

[목차]

Chapter 10. 텐서플로에서 데이터 적재와 전처리하기

13.2 TFRecord 포맷

13.3 입력 특성 전처리

13.4 TF 변환

13.5 텐서플로 데이터셋(TFDS) 프로젝트

13.6 연습문제

배
우
내
용

13.2 TFRecord 포맷

TFRecord 포맷

- tensorflow 는 대용량 데이터를 저장하기 위해 tfrecord 라는 포맷을 사용한다.
- tfrecord 는 크기가 다른 여러가지 레코드를 저장하는 이진 포맷이다.
- 각 레코드는 레코드 길이 CRC checksum(길이가 올바른지 체크하는), 실제 데이터, 데이터를 위한 CRC checksum 으로 구성된다.

```
with tf.io.TFRecordWriter("my_data.tfrecord") as f:
    f.write(b"This is the first record")
    f.write(b"And this is the second record")
```

- 위 코드와 같은 방법으로 tfrecord 를 작성할 수 있다.

```
filepaths = ["my_data.tfrecord"]
dataset = tf.data.TFRecordDataset(filepaths)
for item in dataset:
    print(item)

#tf.Tensor(b'This is the first record', shape=(), dtype=string)
#tf.Tensor(b'And this is the second record', shape=(), dtype=string)
```

- tfrecord 를 불러올 때는 filepath 를 인자로 사용해 `tf.data.TFRecordDataset()` 을 통해 불러올 수 있다.

```
filepaths = ["my_test_{}.tfrecord".format(i) for i in range(5)]
for i, filepath in enumerate(filepaths):
    with tf.io.TFRecordWriter(filepath) as f:
        for j in range(3):
            f.write("File {} record {}".format(i, j).encode("utf-8"))

dataset = tf.data.TFRecordDataset(filepaths, num_parallel_reads=3)
for item in dataset:
    print(item)
```

- `list_files()`와 `interleave()`를 사용했던 것 처럼 여러 파일에서 레코드를 위 코드처럼 번갈아가며 읽을 수도 있다.

13.2.1 압축된 TFRecord 파일

- tfrecord file 을 압축하여 저장할 수 있다.

```
options = tf.io.TFRecordOptions(compression_type="GZIP")
with tf.io.TFRecordWriter("my_compressed.tfrecord", options) as f:
    f.write(b"This is the first record")
    f.write(b"And this is the second record")
```

- 저장하는 코드이다. options 만 지정해주면 압축이 가능하다.

```
dataset = tf.data.TFRecordDataset(["my_compressed.tfrecord"],
                                   compression_type="GZIP")

for item in dataset:
    print(item)
```

13.2.2 프로토콜 버퍼 개요

프로토콜 버퍼

직렬화

- 데이터를 파일로 저장하거나 네트워크 통신이 가능하도록 형식을 바꾸어 주는 것.

프로토콜 버퍼

- 프로토콜 버퍼는 google 이 개발한 이진 포맷으로 파일 저장이나 네트워크 전송 등을 위해 사용한다.
- 직렬화 된 데이터를 이진 포맷으로 저장하기 때문에 더 적은 용량으로 데이터 전송이 가능하다.
- 데이터를 직렬화 하기 때문에 Language Neutral 하다.
- jpg, png 파일과 같은 이미지 파일들을 사용할 때 필요한 인코딩, 디코딩 작업이 필요 없이 직렬화된 데이터를 읽으면 되므로 편리하다.
- 보통 데이터를 보관할 때 data, target 을 분리하여 보관하게 되는데, 프로토콜 버퍼를 사용하면 직렬화 하여 하나의 파일로 보관할 수 있기 때문에 data 와 target 을 매칭해주는 코드를 추가적으로 작성할 필요가 없어져 불필요한 코드를 줄일 수 있다.

프로토콜 버퍼 생성

```
#person.proto로 파일 저장
%%writefile person.proto
syntax = "proto3"; #protocol buffer version3
message Person {
    string name = 1;
    int32 id = 2;
    repeated string email = 3;
}
```

- C 언어의 구조체 형식과 비슷하게 프로토콜 버퍼를 만들 수 있다.

- 1, 2, 3 은 각각 필드 식별자로 데이터의 이진 표현에 사용된다.

프로토콜 버퍼 컴파일

```
!protoc person.proto --python_out=. --descriptor_set_out=person.desc --include_imports
```

- 프로토콜 버퍼는 protoc 라는 c 언어 기반 컴파일러를 통해 컴파일을 진행하고 --python_out=. 옵션을 통해 python 클래스를 생성해야 사용할 수 있다.

```
!ls
```

```
# person.desc person_pb2.py person.proto
```

- 컴파일이 끝나면 디렉토리에 person.desc, person_pb2.py 2 개의 파일이 추가된다. 이중 pb2(protocol buffer 2)가 붙은 파일을 import 하여 클래스를 사용할 수 있다.

```
from person_pb2 import Person

person=Person(name='AI', id=123, email=['a@b.com'])
person.name='Alice' #필드는 수정 가능하다.
person.email.append('c@d.com')
s = person.SerializeToString()
s
#b'\n\x05Alice\x10{\x1a\x07a@b.com\x1a\x07c@d.com'
```

- 프로토콜 버퍼를 사용한 Person 클래스를 사용해 person 객체를 만들었다.
- person 객체의 각 필드들은 수정이 가능하고, 반복 필드(배열)의 경우 인덱싱을 통해 참조 또한 가능하다.
- person 객체는 ParseFromString() 메소드를 통해 직렬화 할 수 있다.
- 직렬화 한 데이터는 네트워크를 통해 전송이 가능하다.

```
from person_pb2 import Person
person2=Person() #객체 생성
person2.ParseFromString(s) #파싱
```

- 네트워크를 통해 전송받은 직렬화된 데이터는 ParseFromString() 메소드를 통해 파싱이 가능하다.

- 혹은 직렬화된 데이터를 TFRecord 파일로 저장한 후 읽고 파싱하는 것도 가능하다.
- 하지만, `SerializeToString()` 과 `ParseFromString()` 은 텐서플로우 연산이 아니기 때문에 텐서플로우 함수에 포함할 수 없다.
- 텐서플로우에서는 이러한 문제를 해결하기 위해 특별한 프로토콜 버퍼 정의를 가지고 있다. 이를 13.2.3 에서 살펴본다.

13.2 텐서플로 프로토콜 버퍼

- TFRecord 파일에서 주로 사용하는 프로토콜 버퍼는 Example 프로토콜 버퍼이다.

```
syntax = "proto3";

message ByteList { repeated bytes value = 1; }
message FloatList { repeated float value = 1 [packed = true]; }
message Int64List { repeated int64 value = 1 [packed = true]; }
message Feature {
  oneof kind {
    ByteList bytes_list = 1;
    FloatList float_list = 2;
    Int64List int64_list = 3;
  }
};

message Features { map<string, Feature> feature = 1; };
message Example { Features features = 1; };
```

- Example 프로토콜 버퍼의 구조는 위와 같다.

Example 클래스를 사용해 객체 생성

```

from tensorflow.train import ByteList, FloatList, Int64List
from tensorflow.train import Feature, Features, Example

person_example=Example(
    features=Features(
        feature={
            "name" : Feature(bytes_list=ByteList(value=[b"Alice"])),
            "id" : Feature(int64_list=Int64List(value=[1,2,3])),
            "emails" : Feature(bytes_list=ByteList(value=[b"a@b.com",
                                                            b"c@d.com"])))
        }
    )
)

```

데이터 직렬화와 TFRecord 형식으로 저장

```

with tf.io.TFRecordWrite("my_contacts.tfrecord") as f:
    f.write(person_example.SerializeToString())

```

13.2.3 Example 프로토콜 버퍼를 읽고 파싱하기

```

feature_description = {
    "name": tf.io.FixedLenFeature([], tf.string, default_value=""),
    "id": tf.io.FixedLenFeature([], tf.int64, default_value=0),
    "emails": tf.io.VarLenFeature(tf.string),
}

for serialized_example in tf.data.TFRecordDataset(["my_contacts.tfrecord"]):
    parsed_example = tf.io.parse_single_example(serialized_example,
                                                feature_description)

```

- Example 프로토콜 버퍼를 파싱하기
위해서는 `parse_single_example()` 메소드를 사용하여야 한다.
- Example 프로토콜 버퍼를 읽기 위해서는 feature description 을
정의해서 `parse_single_example()` 메소드의 인자로 넣어줘야
한다.

```
dataset=tf.data.TFRecordDataset(["my_contacts.tfrecord"]).batch(10)
for serialized_examples in dataset:
    parsed_examples=tf.io.parse_example(serialized_examples, feature_description)
```

- `parse_exmple()` 메소드를 사용하면 데이터를 하나씩 파싱하는 것이 아니라 배치 단위로 파싱하는 것이 가능해진다.

13.2.4 SequenceExample 프로토콜 버퍼를 사용해 리스트의 리스트 다루기

SequenceExample 프로토콜 버퍼의 정의

```
message FeatureList { repeated Feature feature=1;};
message FeatureLists { map<string, FeatureList> feature_list=1;};
message SequenceExample{
    Feature context = 1;
    FeatureLists feature_lists=2;

}
```

- Features 객체는 문맥 데이터를 정의한다.
- FeatureLists 에는 한 개 이상의 FeatureList 가 포함된다.
- Feature 객체는 바이트 스트링의 리스트나 64 비트 정수의 리스트, 실수의 리스트일 수 있다.

```
parsed_context, parsed_feature_lists=tf.io.parse_single_sequence_example(
    serialized_sequence_example, xontext_feature_descriptions,
    sequence_feature_descriptions)
parsed_context=tf.RaggedTensor.from_sparse(parsed_feature_lists["context"])
```

- Sequence 프로토콜 버퍼를 파싱하는 방법은 Example 프로토콜 버퍼를 파싱하는 방법과 동일하다.

13.3 입력 특성 전처리

데이터 전처리 방법

- 1) numpy, pandas 등을 통해 데이터를 사용하기 전에 전처리
- 2) 데이터 API로 데이터를 적재할 때 동적으로 전처리
- 3) 전처리층을 직접 모델에 포함시킴

이 중에 세 번째 방법 : 전처리층을 직접 모델에 포함시키는 법을 알아보겠다.

```
means=np.mean(X_train, axis=0, keepdims=True)
stds=np.std(X_train, axis=0, keepdims=True)
eps=keras.backend.epsilon()
model=keras.models.Sequential([
    keras.layers.Lambda(lambda inputs: (inputs - means) / (stds + eps)),
    ...
])
```

- keras의 sequential 모델을 생성하여 Lambda layer를 생성하여 전처리를 실행하는 모델이다.
- Lambda layer에서는 standardization을 시행한다.

```
class Standardization(keras.layers.Layer):
    def adapt(self, data_sample):
        self.means_=np.mean(data_sample, axis=0, keepdims=True)
        self.stds_=np.std(data_sample, axis=0, keepdims=True)
    def call(self, inputs):
        return (inputs-self.means_) / (self.stds_ + keras.backend.epsilon())

std_layer=Standardization()
std_layer.adapt(data_sample)

model=keras.Sequential()
model.add(std_layer)
...
model.compile([...])
model.fit([...])
```

- 사용자 정의 클래스를 통해서도 전처리 layer를 생성할 수 있다.
- layer를 사용하기 전에 adapt 메소드를 사용해 데이터셋의 mean, std를 미리 계산해야 한다.

- 이 때 인자로 전체 데이터를 줄 필요 없이 랜덤하게 선택된 수백개의 데이터를 넘겨줘도 충분하다.
- 우리가 생성한 사용자 정의 layer 와 비슷하게 `keras.layers.Normalization()` 메소드가 존재한다. 이를 통해 전처리 층을 쉽게 생성 가능하다.

13.3.1 원-핫 벡터를 사용해 범주형 특성 인코딩하기

- 범주형 특성의 경우 모델 생성을 위해 수치형으로 변환해야 한다.
- 범주형 특성은 one-hot vector 를 통해 수치형으로 인코딩할 수 있다.

```

vocab=["<1H OCEAN", "INLAND", "NEAR OCEAN", "NEAR BAY", "ISLAND"]
indices=tf.range(len(vocab), dtype=tf.int64)
table_init=tf.lookup.KeyValueTensorInitializer(vocab, indices)
num_oov_buckets=2
table=tf.lookup.StaticVocabularyTable(table_init, num_oov_buckets)

categories=tf.constant(["NEAR BAY", "DESERT", "INLAND", "INLAND"])
cat_indices=table.lookup(categories)
cat_one_hot=tf.one_hot(cat_indices, depth=len(vocab) + num_oov_buckets)
cat_one_hot

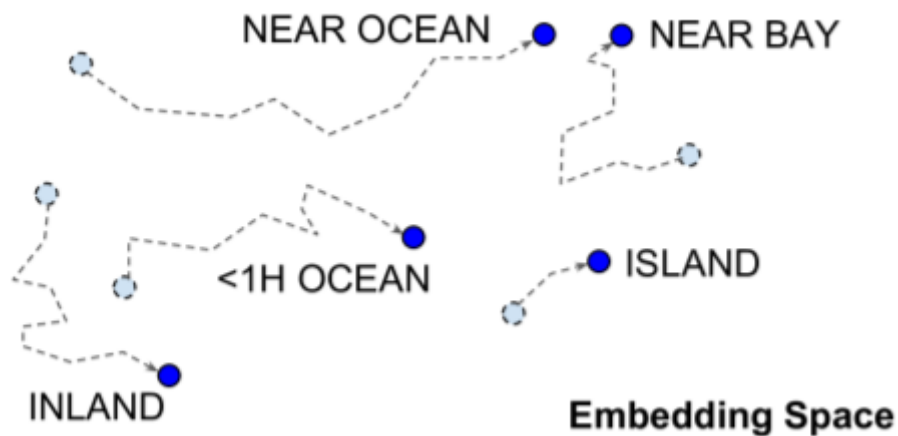
# <tf.Tensor: shape=(4, 7), dtype=float32, numpy=
# array([[0., 0., 0., 1., 0., 0., 0.],
#        [0., 0., 0., 0., 0., 1., 0.],
#        [0., 1., 0., 0., 0., 0., 0.],
#        [0., 1., 0., 0., 0., 0., 0.]], dtype=float32)>

```

- oov bucket 은 vocab 에 정의되지 않은 범주를 대비하여 생성한다. 범주 개수가 너무 많아 사전에 정의해놓기 어려울 경우 oov bucket 을 활용한다.
- categories 중 "DESERT"가 정의되지 않은 범주이다. 이 범주는 one-hot encoding 시 oov bucket 의 위치인 5, 6 번 중 5 번에 매핑된다.

- 케라스 API 에는 동일한 작업을 수행하는 `keras.layers.TextVectorization` 층이 포함되어 있다. `adapt()` 이에 관해 추후에 연습문제에서 살펴본다.
- 범주가 몇 개 되지 않을 경우엔 one-hot encoding 을 사용한다. 하지만, 범주 개수가 50 개 이상이면 embedding 이 선호된다. 10~50 개 사이에 있다면 두 개를 모두 사용해 보고 최적 기법을 찾아 적용하면 된다.

13.3.2 임베딩을 사용해 범주형 특성 인코딩하기



- 임베딩은 범주를 표현하는 훈련 가능한 밀집 벡터를 뜻한다.
- 임베딩 값은 초기에 랜덤으로 초기화되고, 벡터의 차원 수는 하이퍼 파라미터를 통해 지정이 가능하다.
- 비슷한 의미를 가진 단어 벡터들 간의 거리는 가까워지고 반대의 경우엔 멀어진다.

```

vocab=["<1H OCEAN", "INLAND", "NEAR OCEAN", "NEAR BAY", "ISLAND"]
indices=tf.range(len(vocab), dtype=tf.int64)
table_init=tf.lookup.KeyValueTensorInitializer(vocab, indices)
num_oov_buckets=2
table=tf.lookup.StaticVocabularyTable(table_init, num_oov_buckets)

categories=tf.constant(["NEAR BAY", "DESERT", "INLAND", "INLAND"])
cat_indices=table.lookup(categories)
tf.nn.embedding_lookup(embedding_matrix, cat_indices)

```

```

# <tf.Tensor: shape=(4, 2), dtype=float32, numpy=
# array([[0.7309859, 0.28189003],
#        [0.6422187, 0.732231 ],
#        [0.05060315, 0.41399097],
#        [0.05060315, 0.41399097]], dtype=float32)>

```

- `tf.nn.embedding_lookup()` 함수는 임베딩 행렬에서 주어진 인덱스에 해당하는 행을 찾는다.

```

embedding=keras.layers.Embedding(input_dim=len(vocab) + num_oov_buckets,
                                  output_dim=embedding_dim)

embedding(cat_indices)

```

```

# <tf.Tensor: shape=(4, 2), dtype=float32, numpy=
# array([[ 0.03839845,  0.01719275],
#        [ 0.01920623,  0.03352095],
#        [-0.03839232, -0.04356638],
#        [-0.03839232, -0.04356638]], dtype=float32)>

```

- 케라스에는 이러한 임베딩 행렬을 처리해주는 `keras.layers.Embedding` 층이 존재한다. 이를 사용하면 쉽게 Embedding layer 를 구현할 수 있다.
- 층이 생성될 때 embedding matrix 를 random 하게 초기화 하고 어떤 범주 인덱스로 호출될 때 임베딩 행렬에 있는 인덱스의 행을 반환한다.

```
regular_inputs=keras.layers.Input(shape=[8])
categories=keras.layers.Input(shape=[], dtype=tf.string)
cat_indices=keras.layers.Lambda(lambda cats: table.lookup(cats))(categories)
cat_embed=keras.layers.Embedding(input_dim=6, output_dim=2)(cat_indices)
encoded_inputs=keras.layers.concatenate([regular_inputs, cat_embed])
outputs=keras.layers.Dense(1)(encoded_inputs)
model=keras.models.Model(inputs=[regular_inputs, categories],
                          outputs=[outputs])
```

- embedding layer 를 사용하여 케라스 모델을 생성한 코드이다.

13.3.3 케라스 전처리 층

`keras.layers.Discretization()`

- Discretization 층은 연속적인 데이터를 특정 개수의 구간으로 나누어 one-hot encoding 을 시행한다. 연속적인 값을 구간으로 나누어 처리하는 것은 잃는 정보가 많은 반면 연속적인 값에서는 관찰할 수 없는 특징이나 패턴을 관찰할 수 있다는 점에서 사용할만 하다.
- Discretization 층은 미분 가능하지 않지만, 미분 가능할 필요가 없다. 전처리 층에 포함되고, 전처리층의 경우 경사하강법에 의해 영향을 받는 층이 아니기 때문이다.

`keras.layers.PreprocessingStage()`

```
normalization=keras.layers.Normalization()
discretization=keras.layers.Discretization([...])
pipeline=keras.layers.PreprocessingStage([normalization, discretization])
pipeline.adapt(data_sample)
```

- 전처리 층을 연결하는 파이프라인을 구성할 때 쓰인다.

이 외에도 여러가지 전처리층이 존재한다.

13.4 TF 변환

데이터 처리

- 데이터가 작은 경우 : `cache()` 메소드를 통해 RAM 에 저장해 놓고 저장된 데이터를 호출하여 사용한다.
- 데이터가 클 경우 : Apache Beam 이나 Spark 같은 도구를 통해 대용량 데이터 처리를 위한 클러스터 컴퓨팅 엔진을 활용하여 pipeline 을 구축한다. 이를 통해 모든 훈련 데이터를 훈련 전에 전처리할 수 있다.

훈련 서빙 왜곡

- 훈련시 전처리 속도와 배포 환경인 앱이나 브라우저의 전처리 속도에 차이가 생기는 것을 뜻한다.
- 사전 전처리의 경우 모델을 여러가지 플랫폼으로 배포했을 때 전처리 코드를 추가해야 하는 번거로움이 생긴다. 유지 보수를 어렵게 만들고, 앱이나 브라우저에서 전처리 연산을 추가적으로 수행해야 하므로 버그나 성능이 감소된다.
- 이를 해결하기 위해 층을 동적으로 추가하는 방법을 사용할 수도 있다. 하지만 이 것도 층에서의 동적인 전처리 연산 과정으로 인해 처리 속도를 감소 시킨다는 단점이 있다.

TF 변환

- TF 변환은 전처리 연산을 한 번만 수행하기 때문에 훈련 서빙 왜곡과 같은 현상이 일어나지 않고, 층을 동적으로 추가할 때의 단점도 상쇄할 수 있다는 장점이 있다.

```
import tensorflow_transform as tft
def preprocess(inputs):
    median_age = inputs["housing_median_age"]
    ocean_proximity = inputs["ocean_proximity"]
    standardized_age = tft.scale_to_z_score(median_age)
    ocean_proximity_id = tft.compute_and_apply_vocabulary(ocean_proximity)
    return {
        "standardized_median_age": standardized_age,
        "ocean_proximity_id": ocean_proximity_id
    }
```

- 아파치 빔을 사용해 이러한 전처리 함수를 전체 훈련 세트에 적용할 수 있다.

13.5 텐서플로 데이터셋 프로젝트

- 텐서플로우 데이터셋은 다양한 데이터셋을 제공한다.

```
import tensorflow_datasets as tfds

dataset=tfds.load(name='mnist')
mnist_train, mnist_test = dataset['train'], dataset['test']

mnist_train=mnist_train.shuffle(10000).batch(32).prefetch(1)
for item in mnist_train:
    images=item['image']
    labels=item['label']
    [...]

mnist_train=mnist_train.shuffle(10000).batch(32)
mnist_train=mnist_train.map(lambda items: (items['image'], items['label']))
mnist_train=mnist_train.prefetch(1)

dataset=tfds.load(name='mnist', batch_size=32, as_supervised=True)
mnist_train=dataset['train'].prefetch(1)
model=keras.models.Sequential([...])
model.compile(loss='sparse_categorical_crossentropy', optimizer='sgd')
model.fit(mnist_train, epochs=5)
```

- 데이터셋에는 각 아이템의 특성과 레이블을 담은 딕셔너리가 있다.
- 데이터셋을 받은 후 map()메서드를 통해 데이터를 변환시켜 사용해야 한다.

13.6 연습문제

1. 왜 데이터 API를 사용해야 하는가?

1. 여러 소스에서 데이터를 읽는 것이 편리하다.

```
1 n_readers=5
2 dataset=filepath_dataset.interleave(
3     lambda filepath: tf.data.TextLineDataset(filepath).skip(1),
4     cycle_length=n_readers)

1 for line in dataset.take(5):
2     print(line.numpy())
```

b'4.5909,16.0,5.475877192982456,1.0964912280701755,1357.0,2.9758771929824563,33.63,-117.71,2.418'
b'2.4792,24.0,3.4547038327526134,1.1341463414634145,2251.0,3.921602787456446,34.18,-118.38,2.0'
b'4.2708,45.0,5.121387283236994,0.953757225433526,492.0,2.8439306358381504,37.48,-122.19,2.67'
b'2.1856,41.0,3.7189873417721517,1.0658227848101265,803.0,2.0329113924050635,32.76,-117.12,1.205'
b'4.1812,52.0,5.701388888888889,0.9965277777777778,692.0,2.4027777777777777,33.73,-118.31,3.215'

- 앞서 13.1에서 여러 개의 filepath들을 하나의 데이터셋 객체로 생성한 후 `interleave()` 메소드를 통해 여러 데이터에서 1줄 씩 데이터를 읽어오는 것이 가능한 것을 확인했다.
- 데이터 API를 사용하지 않았다면, 반복문을 통한 더 복잡한 처리가 필요했을 것이다.

2. 데이터 변환이 용이하다.

```
1 X=tf.range(10)
2 dataset=tf.data.Dataset.from_tensor_slices(X)
3 dataset=dataset.map(lambda x: x * 2)

1 X=tf.range(10)
2 dataset=tf.data.Dataset.from_tensor_slices(X)
3 dataset=dataset.filter(lambda x: x<10)
```

- `map()`, `filter()` 등의 메소드를 통해 객체의 데이터에 변환 처리를 해 줄 수 있다.

3. 데이터를 섞을 수 있다.


```

1 dataset=tf.data.Dataset.range(10).repeat(3)
2 dataset=dataset.shuffle(buffer_size=5, seed=42).batch(7)
3 for item in dataset:
4     print(item)

```

```

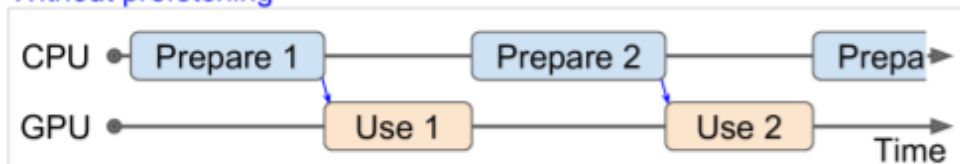
tf.Tensor([0 2 3 6 7 9 4], shape=(7,), dtype=int64)
tf.Tensor([5 0 1 1 8 6 5], shape=(7,), dtype=int64)
tf.Tensor([4 8 7 1 2 3 0], shape=(7,), dtype=int64)
tf.Tensor([5 4 2 7 8 9 9], shape=(7,), dtype=int64)
tf.Tensor([3 6], shape=(2,), dtype=int64)

```

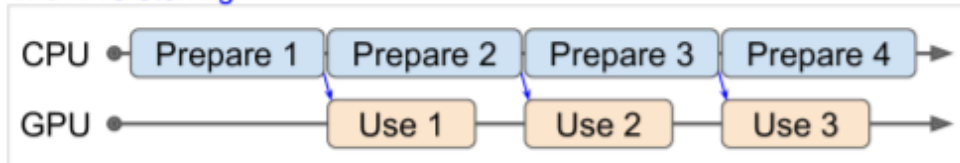
- shuffle() 메소드를 사용하여 데이터를 섞을 수 있다.

4. 프리페치 기능을 사용할 수 있다.

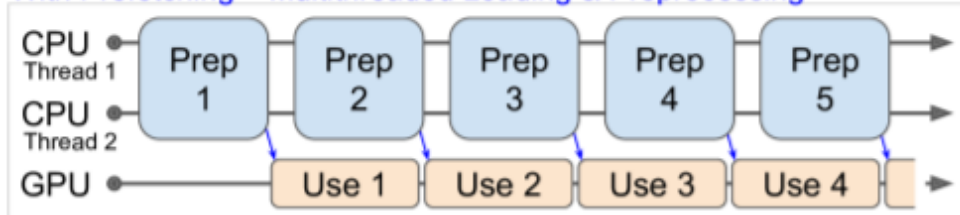
Without prefetching



With Prefetching

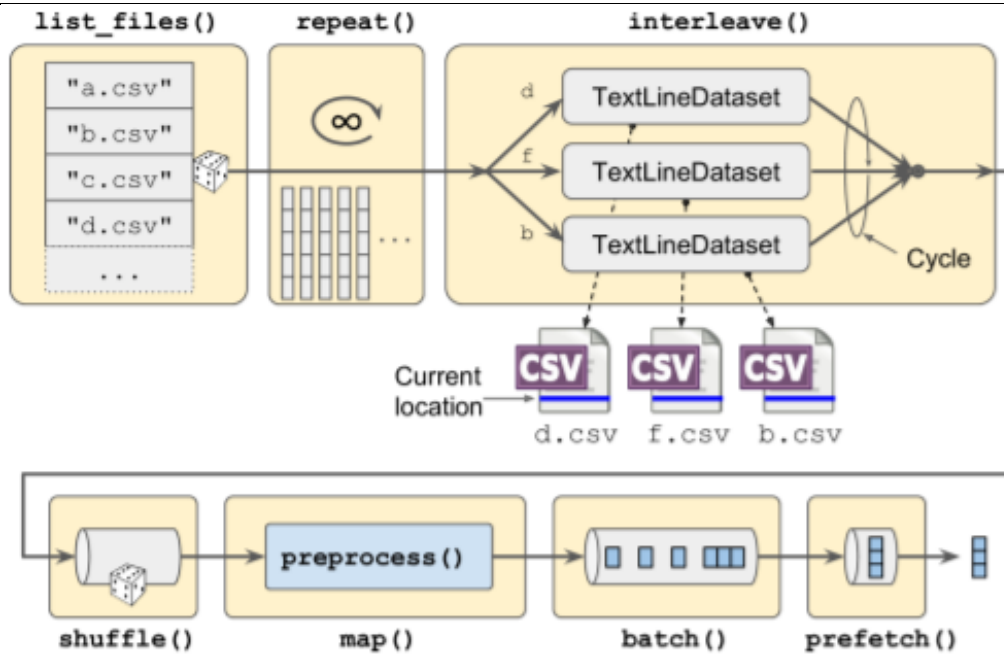


With Prefetching + Multithreaded Loading & Preprocessing



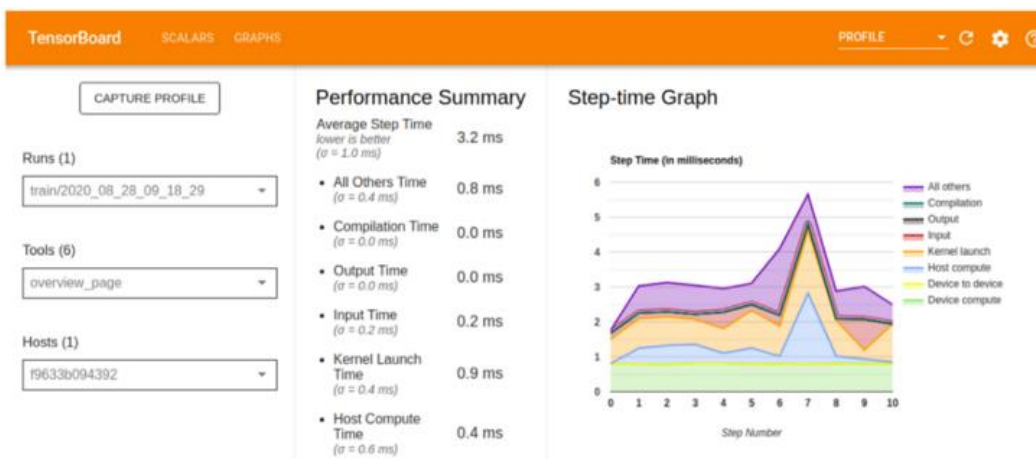
- prefetch() 기능을 통해 학습 시간을 단축시킬 수 있다.

2. 대용량 데이터셋을 여러 파일로 나눌 때 장점



- shuffle()메소드를 사용하여 데이터를 섞기 전에 크게 섞는 것이 가능해짐
- repeat()메소드를 사용하면 내부적으로 file path들이 크게 한 번 섞임.
- 또한 한 대의 컴퓨터로 처리하기 힘든 대용량 파일을 여러 파일로 나누어 여러 서버에 저장하고, 사용시에 여러 서버에서 파일들을 읽어들이어 처리하는 것이 가능해진다.

3. 파이프라인의 병목 찾기, 병목현상 개선 방법



- 병목 현상은 CPU와 GPU의 성능 차이 때문에 모델 훈련 속도가 느려지는 것을 말한다.
- 파이프라인의 병목 현상은 텐서보드의 프로파일러를 사용하여 확인

할 수 있다.

- https://www.tensorflow.org/tensorboard/tensorboard_profiling_keras 참고
- 만약 GPU보다 낮은 성능의 CPU 때문에 병목 현상이 일어나고 있다면, 여러 스레드에서 동시에 데이터를 읽고(iterative()의 num_parallel_calls 하이퍼 파라미터 조정) 전처리하여 몇 개의 배치를 프리페치(prefetch()메소드 사용) 하여 해결할 수 있다. 혹은 원본 데이터를 여러 대의 서버에 여러 개의 파일로 나누어 놓으면 network bandwidth를 효율적으로 활용할 수 있다.

4. 어떤 이진 데이터도 TFRecord파일 또는 직렬화된 프로토콜 버퍼로 저장할 수 있는가?

- TFRecord 파일은 이진 데이터로 이뤄져 있고, 각 레코드에는 원하는 이진 데이터를 저장할 수 있다. 앞서 살펴봤듯이 객체까지도 직렬화 하여 저장할 수 있고 이미지 데이터도 직렬화 하여 저장하는게 가능하다.

5. 모든 데이터를 Example 프로토콜 버퍼 포맷으로 변환해야 하는가? 자신만의 프로토콜 버퍼 정의를 사용하는 것은 어떤가?

- Example 프로토콜 버퍼를 사용하면 텐서플로우가 기본적으로 제공하는 파싱 연산을 사용할 수 있기 때문에 편하다. 또한 추가적인 컴파일 과정이 필요없다.
- 하지만, 사용자 정의 프로토콜을 사용하게 되면 protoc(protocol buffer compiler)를 사용하여 컴파일을 해야 하고, 모델 배포시 프로토콜 버퍼에 대한 정보를 담은 discriptor도 추가적으로 배포해야 하므로 복잡하다는 단점이 있다.

6. TFRecord를 사용할 때 언제 압축을 사용하는가? 왜 기본적으로 압축을 사용하지 않는가?

- TFRecord를 사용할 때 훈련 스크립트로 TFRecord 파일을 다운로드

해야 될 경우 파일 다운로드에 따른 시간을 줄이기 위하여 압축을 활성화 하는게 맞다.

- 하지만, 만약 훈련 스크립트와 같은 머신에 파일이 존재한다면, 압축을 해제하는데 CPU자원을 소모하지 않기 위해 압축하지 않는게 좋다.

7. 데이터 파일을 작성할 때, 또는 tf.data 파이프라인 안에서, 모델의 전처리 층에서, TF변환을 사용하여 데이터를 전처리할 수 있다. 각 방식의 장단점은?

1) 데이터 파일을 작성할 때 전처리

장점

- 훈련 도중에 동적으로 전처리를 할 필요가 없어지기 때문에 거쳐야 될 과정이 줄어들어 훈련 스크립트 자체는 빨리 실행될 것이다.
- 전처리 된 데이터가 원본 데이터보다 훨씬 작아질 경우 공간이 절약 되고 다운로드 속도가 빨라진다. 예를 들어 크롤링을 통해 수집된 이미지 데이터의 경우 각기 다른 해상도를 가지고 있는데, 이를 동일한 size로 resize할 경우 원본 데이터 용량이 훨씬 줄어들 것임.

1) 데이터 파일을 작성할 때 전처리

단점

- 전처리 로직을 변경해가며 좋은 전처리 로직을 찾는 실험을 할 때 , 로직마다 전처리된 데이터셋을 생성해야 한다.
- 데이터 증식(Data Augmentation)을 수행하려면 변환한 데이터들을 생성하고 저장해야 하므로 디스크 공간이 추가적으로 필요하고, 처리를 하는데 드는 시간도 추가적으로 필요하다.
- 모델이 전처리된 데이터를 통해 훈련되었기 때문에 프로그램 내에 전처리 코드를 추가해야 모델이 제대로 작동한다.

2) tf.data 파이프라인을 사용해 데이터를 전처리

장점

- 전처리 로직을 변경하고 데이터 증식을 사용하기 훨씬 쉽다.
- tf.data를 사용하면 효율적인 전처리 파이프라인을 만들 수 있다(멀티

스레딩과 프리페칭을 이용한 GPU활용도 증가).

단점

- 훈련 속도가 느려짐.
- 에포크마다 모델이 데이터 전처리를 해줘야 함

3) 모델의 전처리 층에서 전처리

장점

- 여러 종류의 플랫폼으로 모델을 내보낼 때 전처리 코드를 여러 번 작성할 필요가 없음.

단점

- 훈련 속도가 느려짐.
- 전처리 연산을 CPU가 아닌 GPU에서 실행할 수 밖에 없음. CPU로 여러 스레드를 통해 병렬 처리 하거나, prefetch를 통해 데이터를 미리 준비해 활용하는 것이 불가능함.

8. 범주형 특성을 인코딩할 수 있는 대표적인 방법을 나열해 보아라. 텍스트 데이터는 어떻게 인코딩할 수 있는가?

텍스트 데이터 인코딩 방법

- 텍스트 데이터를 인코딩 하는 가장 대표적인 방법은 BoW(Bag of Words) 표현법이 있다.
- 각 단어들에 정수 인덱스를 부여하고, 각 인덱스의 위치에 단어 토큰의 등장 횟수를 기록한 벡터를 만드는 방법이다.
- 이 때, 중요한 단어들에 집중하기 위해 TF-IDF(Term Frequency - Inverse Document Frequency) 가중치를 부여한다. TF-IDF의 기본 아이디어는 빈도수가 과도하게 높은 단어들(a, the와 같은)에 패널티를 부여하여 중요도를 낮추는 것이다.

-	과일이	길고	노란	먹고	바나나	사과	싫은	저는	좋아요
문서1	0	0	0	1	0	1	1	0	0
문서2	0	0	0	1	1	0	1	0	0
문서3	0	1	1	0	2	0	0	0	0
문서4	1	0	0	0	0	0	0	1	1

-	과일이	길고	노란	먹고	바나나	사과	싫은	저는
문서1	0	0	0	0.287682	0	0.693147	0.287682	0
문서2	0	0	0	0.287682	0.287682	0	0.287682	0
문서3	0	0.693147	0.693147	0	0.575364	0	0	0
문서4	0.693147	0	0	0	0	0	0	0.6931

단어	IDF(역 문서 빈도)
과일이	$\ln(4/(1+1)) = 0.693147$
길고	$\ln(4/(1+1)) = 0.693147$
노란	$\ln(4/(1+1)) = 0.693147$
먹고	$\ln(4/(2+1)) = 0.287682$
바나나	$\ln(4/(2+1)) = 0.287682$
사과	$\ln(4/(1+1)) = 0.693147$
싫은	$\ln(4/(2+1)) = 0.287682$
저는	$\ln(4/(1+1)) = 0.693147$
좋아요	$\ln(4/(1+1)) = 0.693147$

- $tf(d, t)$: 특정 문서 d에서 특정 단어 t의 등장 횟수
- $df(t)$: 특정 단어 t가 등장한 문서의 수

- $idf(d, t) : df(t)$ 에 반비례하는 수
- $idf(d, f) = \log\left(\frac{n}{1+df(t)}\right)$
- tf에 idf를 곱해주면 tf-idf 가중치를 적용한 데이터가 된다.

9번. tf.data api를 활용해 전처리층을 포함한 케라스 모델 만들기

데이터셋 로드 및 프로토콜 버퍼 변환 함수 생성

```
1 import tensorflow as tf
2 from tensorflow import keras
3 from tensorflow.train import BytesList, FloatList, Int64List
4 from tensorflow.train import Feature, Features, Example
5 import matplotlib.pyplot as plt
6 import numpy as np
7 import os

1 #fashion mnist data load and train_test_split
2 (X_train_full, y_train_full), (X_test, y_test) = keras.datasets.fashion_mnist.load_data()
3 X_valid, X_train = X_train_full[:5000], X_train_full[5000:]
4 y_valid, y_train = y_train_full[:5000], y_train_full[5000:]

1 #train set tf.data 객체 생성
2 train_set = tf.data.Dataset.from_tensor_slices((X_train, y_train)).shuffle(len(X_train))
3 valid_set = tf.data.Dataset.from_tensor_slices((X_valid, y_valid))
4 test_set = tf.data.Dataset.from_tensor_slices((X_test, y_test))

1 #각 레코드(이미지)를 Example protocol buffer로 변환하는 함수
2 def create_example(image, label):
3     image_data = tf.io.serialize_tensor(image)
4     return Example(
5         features=Features(
6             feature={
7                 "image": Feature(bytes_list=BytesList(value=[image_data.numpy()])),
8                 "label": Feature(int64_list=Int64List(value=[label])),
9             })
10     )
```

원본 data를 여러 개의 file로 분할하는 함수 생성 및 데이터 분할

```

1 #여러개의 file들로 dataset 분할, tfrecord형식으로 저장
2 from contextlib import ExitStack
3
4 def write_tfrecords(name, dataset, n_shards=10):
5     paths = [("{}tfrecord-{:05d}-of-{:05d}").format(name, index, n_shards)
6               for index in range(n_shards)] # path형식
7     with ExitStack() as stack:
8         writers = [stack.enter_context(tf.io.TFRecordWriter(path))
9                     for path in paths]
10        for index, (image, label) in dataset.enumerate():
11            shard = index % n_shards
12            example = create_example(image, label)
13            writers[shard].write(example.SerializeToString())
14    return paths

```

```

1 #write_tfrecords 함수를 통해 데이터 분할
2 train_filepaths = write_tfrecords("my_fashion_mnist.train", train_set)
3 valid_filepaths = write_tfrecords("my_fashion_mnist.valid", valid_set)
4 test_filepaths = write_tfrecords("my_fashion_mnist.test", test_set)

```

전처리 함수와 데이터셋 생성 함수 정의 및 데이터셋 생성

```

1 #전처리 함수
2 def preprocess(tfrecord):
3     feature_descriptions = {
4         "image": tf.io.FixedLenFeature([], tf.string, default_value=""),
5         "label": tf.io.FixedLenFeature([], tf.int64, default_value=-1)
6     }
7     example = tf.io.parse_single_example(tfrecord, feature_descriptions)
8     image = tf.io.parse_tensor(example["image"], out_type=tf.uint8)
9     image = tf.reshape(image, shape=[28, 28]) #reshape
10    return image, example["label"]
11
12 # filepaths로 dataset 생성 함수
13 def mnist_dataset(filepaths, n_read_threads=5, shuffle_buffer_size=None,
14                   n_parse_threads=5, batch_size=32, cache=True):
15     #TFRecordDataset으로 저장된 데이터를 dataset에 초기화
16     dataset = tf.data.TFRecordDataset(filepaths,
17                                       num_parallel_reads=n_read_threads) #멀티스레드
18     if cache:
19         dataset = dataset.cache() #메모리에 캐시
20     if shuffle_buffer_size: #shuffle값을 주면 조건문 작동
21         dataset = dataset.shuffle(shuffle_buffer_size) #shuffle
22     dataset = dataset.map(preprocess, num_parallel_calls=n_parse_threads)
23     dataset = dataset.batch(batch_size)
24     return dataset.prefetch(1) #prefetch로 효율 높임

```

```

1 # mnist_dataset함수로 dataset 생성
2 train_set = mnist_dataset(train_filepaths, shuffle_buffer_size=60000)
3 valid_set = mnist_dataset(valid_filepaths)
4 test_set = mnist_dataset(test_filepaths)

```

사용자 정의 클래스를 통해 standardization class를 정의, 모델에 layer로서 적용


```

1 #사용자 정의 함수를 통해 standardization class 생성
2 class Standardization(keras.layers.Layer):
3     def adapt(self, data_sample):
4         self.means_ = np.mean(data_sample, axis=0, keepdims=True)
5         self.stds_ = np.std(data_sample, axis=0, keepdims=True)
6     def call(self, inputs):
7         return (inputs - self.means_) / (self.stds_ + keras.backend.epsilon())
8
9 standardization = Standardization(input_shape=[28, 28])
10
11 #전체 데이터를 다 사용할 필요 없이 100개만 random으로 뽑음
12 sample_image_batches = train_set.take(100).map(lambda image, label: image)
13 #sample image concatenate
14 sample_images = np.concatenate(list(sample_image_batches.as_numpy_iterator()),
15                                axis=0).astype(np.float32)
16 #adapt 메소드를 통해 데이터의 mean, std를 구함
17 standardization.adapt(sample_images)
18
19 #standardization layer를 사용해 model의 층에서 전처리를 하는 모델 생성
20 model = keras.models.Sequential([
21     standardization,
22     keras.layers.Flatten(),
23     keras.layers.Dense(100, activation="relu"),
24     keras.layers.Dense(10, activation="softmax")
25 ])
26 model.compile(loss="sparse_categorical_crossentropy",
27               optimizer="nadam", metrics=["accuracy"])

```

생성한 모델로 훈련

```

1 from datetime import datetime
2 logs = os.path.join(os.getcwd(), "my_logs",
3                     "run_" + datetime.now().strftime("%Y%m%d_%H%M%S"))
4
5 tensorboard_cb = tf.keras.callbacks.TensorBoard(
6     log_dir=logs, histogram_freq=1, profile_batch=10)
7
8 model.fit(train_set, epochs=50, validation_data=valid_set,
9         callbacks=[tensorboard_cb])

```

Epoch 1/50
1719/1719 [=====] - 7s 4ms/step - loss: 55.0552 - accuracy: 0.9133 - val_loss: 0.1660
Epoch 2/50
1719/1719 [=====] - 6s 4ms/step - loss: 211.0661 - accuracy: 0.9187 - val_loss: 0.1590
Epoch 3/50
1719/1719 [=====] - 7s 4ms/step - loss: 85.6928 - accuracy: 0.9239 - val_loss: 0.1590
Epoch 4/50
1719/1719 [=====] - 6s 4ms/step - loss: 73.9704 - accuracy: 0.9276 - val_loss: 0.1590
Epoch 5/50
1719/1719 [=====] - 7s 4ms/step - loss: 104.1989 - accuracy: 0.9312 - val_loss: 0.1590
Epoch 6/50
1719/1719 [=====] - 7s 4ms/step - loss: 13.4018 - accuracy: 0.9362 - val_loss: 0.1590
Epoch 7/50
1719/1719 [=====] - 7s 4ms/step - loss: 0.1660 - accuracy: 0.9397 - val_loss: 0.1590
Epoch 8/50
1719/1719 [=====] - 7s 4ms/step - loss: 0.1590 - accuracy: 0.9416 - val_loss: 0.1590
Epoch 9/50

10번(a~d). imbd 데이터셋을 통해 Textvectorization layer와 BoW layer 를 적용한 model 만들기

imdb 데이터 로드 및 계층 구조 확인

```
1 from pathlib import Path
2 import numpy as np
3 import os

1 #imdb 데이터 로드
2 DOWNLOAD_ROOT = "http://ai.stanford.edu/~amaas/data/sentiment/"
3 FILENAME = "aclImdb_v1.tar.gz"
4 filepath = keras.utils.get_file(FILENAME, DOWNLOAD_ROOT + FILENAME, extract=True)
5 path = Path(filepath).parent / "aclImdb"
6 path

Downloading data from http://ai.stanford.edu/~amaas/data/sentiment/aclImdb_v1.tar.gz
84131840/84125825 [=====] - 3s 0us/step
84140032/84125825 [=====] - 3s 0us/step
PosixPath('/root/.keras/datasets/aclImdb')

1 # os.walk 함수를 통해 top-down 방식으로 sub directory와 모든 파일 출력
2 for name, subdirs, files in os.walk(path):
3     indent = len(Path(name).parts) - len(path.parts)
4     print("    " * indent + Path(name).parts[-1] + os.sep)
5     for index, filename in enumerate(sorted(files)):
6         #인덱스 3 이상부터는 ...로 표기하여 생략
7         if index == 3:
8             print("    " * (indent + 1) + "...")
9             break
10        print("    " * (indent + 1) + filename)
```

데이터 변수 할당

```
1 def review_paths(dirpath):
2     #dirpath에서 .txt로 끝나는 모든 파일을 return
3     return [str(path) for path in dirpath.glob("*.txt")]
4
5 #path = '/root/.keras/datasets/aclImdb'
6 #review_paths함수를 통해 train, test_valid로 구분해 positive, negative 별로 데이터셋 생성
7 train_pos = review_paths(path / "train" / "pos")
8 train_neg = review_paths(path / "train" / "neg")
9 test_valid_pos = review_paths(path / "test" / "pos")
10 test_valid_neg = review_paths(path / "test" / "neg")
11
12 len(train_pos), len(train_neg), len(test_valid_pos), len(test_valid_neg)

1 #test_valid를 test, valid로 분할
2 np.random.shuffle(test_valid_pos)
3
4 test_pos = test_valid_pos[:5000]
5 test_neg = test_valid_neg[:5000]
6 valid_pos = test_valid_pos[5000:]
7 valid_neg = test_valid_neg[5000:]
8
9 len(test_pos), len(test_neg), len(valid_pos), len(valid_neg)
```

tf.data를 통해 데이터셋 생성

```

1 #tf.data를 통해서 데이터 적재
2 def imdb_dataset(filepaths_positive, filepaths_negative, n_read_threads=5):
3     dataset_neg = tf.data.TextLineDataset(filepaths_negative,
4                                           num_parallel_reads=n_read_threads)
5     dataset_neg = dataset_neg.map(lambda review: (review, 0))
6     dataset_pos = tf.data.TextLineDataset(filepaths_positive,
7                                           num_parallel_reads=n_read_threads)
8     dataset_pos = dataset_pos.map(lambda review: (review, 1))
9     return tf.data.Dataset.concatenate(dataset_pos, dataset_neg)

1 batch_size = 32
2
3 train_set = imdb_dataset(train_pos, train_neg).shuffle(25000).batch(batch_size).prefetch(1)
4 valid_set = imdb_dataset(valid_pos, valid_neg).batch(batch_size).prefetch(1)
5 test_set = imdb_dataset(test_pos, test_neg).batch(batch_size).prefetch(1)

```

Textvectorization layer 정의

```

1 #단어 자르고 소문자로 전처리하는 함수 생성
2 def preprocess(X_batch, n_words=50):
3     shape = tf.shape(X_batch) + tf.constant([1, 0]) + tf.constant([0, n_words])
4     Z = tf.strings.substr(X_batch, 0, 300) #단어 자름
5     Z = tf.strings.lower(Z) #소문자 변환
6     Z = tf.strings.regex_replace(Z, b"<br#\s+/?>", b" ")
7     Z = tf.strings.regex_replace(Z, b"^[a-z]", b" ")
8     Z = tf.strings.split(Z)
9     return Z.to_tensor(shape=shape, default_value=b"<pad>") #default value로 b"<pad>"
10
11 X_example = tf.constant(["It's a great, great movie! I loved it.", "It was terrible, run away!!!"])
12 preprocess(X_example)

1 from collections import Counter
2
3 #preprocess함수를 입력으로 받아 vocabulary return하는 함수
4 def get_vocabulary(data_sample, max_size=1000):
5     preprocessed_reviews = preprocess(data_sample).numpy()
6     counter = Counter()
7     for words in preprocessed_reviews:
8         for word in words:
9             if word != b"<pad>":
10                 counter[word] += 1
11     #[b"<pad>"]+가장 많은 원소 개수부터 내림차순
12     return [b"<pad>"] + [word for word, count in counter.most_common(max_size)]
13
14 #바로 위에서 살펴봤던 X_example을 통해 함수 작동 확인
15 get_vocabulary(X_example)

1 #TextVectorization 클래스 생성
2 class TextVectorization(keras.layers.Layer):
3     def __init__(self, max_vocabulary_size=1000, n_oov_buckets=100, dtype=tf.string, **kwargs):
4         super().__init__(dtype=dtype, **kwargs)
5         self.max_vocabulary_size = max_vocabulary_size
6         self.n_oov_buckets = n_oov_buckets
7
8     def adapt(self, data_sample):
9         self.vocab = get_vocabulary(data_sample, self.max_vocabulary_size) #get vocabulary로 단어 시
10         words = tf.constant(self.vocab) #tensor변환
11         word_ids = tf.range(len(self.vocab), dtype=tf.int64)
12         vocab_init = tf.lookup.KeyValueTensorInitializer(words, word_ids)
13         self.table = tf.lookup.StaticVocabularyTable(vocab_init, self.n_oov_buckets)
14
15     def call(self, inputs):
16         preprocessed_inputs = preprocess(inputs)
17         return self.table.lookup(preprocessed_inputs)

```

```

1 #imdb를 통해 클래스 초기화
2 max_vocabulary_size = 1000
3 n_oov_buckets = 100
4
5 sample_review_batches = train_set.map(lambda review, label: review)
6 sample_reviews = np.concatenate(list(sample_review_batches.as_numpy_iterator()),
7                                 axis=0)
8
9 text_vectorization = TextVectorization(max_vocabulary_size, n_oov_buckets,
10                                     input_shape=[])
11 text_vectorization.adapt(sample_reviews)

```

BoW layer 정의

```

1 #레코드(리뷰 1개)에서 단어가 등장하는 횟수를 count하는 층
2 class BagOfWords(keras.layers.Layer):
3     def __init__(self, n_tokens, dtype=tf.int32, **kwargs):
4         super().__init__(dtype=dtype, **kwargs)
5         self.n_tokens = n_tokens
6     def call(self, inputs):
7         one_hot = tf.one_hot(inputs, self.n_tokens)
8         return tf.reduce_sum(one_hot, axis=1)[:, 1:]

1 #simple_example을 통해 잘 작동하는지 test
2 simple_example = tf.constant([[1, 3, 1, 0, 0], [2, 2, 0, 0, 0]])
3 bag_of_words = BagOfWords(n_tokens=4)
4 bag_of_words(simple_example)

<tf.Tensor: shape=(2, 3), dtype=float32, numpy=
array([[2., 0., 1.],
       [0., 2., 0.]], dtype=float32)>

1 #token 개수를 전달해 bag_of_words 객체 생성
2 n_tokens = max_vocabulary_size + n_oov_buckets + 1 # add 1 for <pad>
3 bag_of_words = BagOfWords(n_tokens)

```

모델 생성 및 훈련

```

1 #token 개수를 전달해 bag_of_words 객체 생성
2 n_tokens = max_vocabulary_size + n_oov_buckets + 1 # add 1 for <pad>
3 bag_of_words = BagOfWords(n_tokens)

```

```

1 model = keras.models.Sequential([
2     text_vectorization,
3     bag_of_words,
4     keras.layers.Dense(100, activation="relu"),
5     keras.layers.Dense(1, activation="sigmoid"),
6 ])
7 model.compile(loss="binary_crossentropy", optimizer="nadam",
8               metrics=["accuracy"])
9 model.fit(train_set, epochs=5, validation_data=valid_set)

```

```

Epoch 1/5
782/782 [=====] - 24s 18ms/step - loss: 0.5441 - accuracy: 0.7127 - val_loss: 0
Epoch 2/5
782/782 [=====] - 21s 21ms/step - loss: 0.4703 - accuracy: 0.7714 - val_loss: 0
Epoch 3/5
782/782 [=====] - 16s 17ms/step - loss: 0.4228 - accuracy: 0.8009 - val_loss: 0
Epoch 4/5
782/782 [=====] - 16s 17ms/step - loss: 0.3570 - accuracy: 0.8445 - val_loss: 0
Epoch 5/5
782/782 [=====] - 16s 17ms/step - loss: 0.2778 - accuracy: 0.8924 - val_loss: 0
<keras.callbacks.History at 0x7f56335ecb50>

```

10-e . Embedding layer를 추가하고 단어 개수의 제곱근을 곱하여 리뷰마다 평균 임베딩을 계산

```

1 # s/sqrt(N)을 return 하는 함수 생성
2 def compute_mean_embedding(inputs):
3     not_pad = tf.math.count_nonzero(inputs, axis=-1)#input의 nonzero 개수
4     n_words = tf.math.count_nonzero(not_pad, axis=-1, keepdims=True)#not_pad의 nonzero개수
5     sqrt_n_words = tf.math.sqrt(tf.cast(n_words, tf.float32))#
6     return tf.reduce_sum(inputs, axis=1) / sqrt_n_words
7
8 another_example = tf.constant([[[1., 2., 3.], [4., 5., 0.], [0., 0., 0.]],
9                                [[6., 0., 0.], [0., 0., 0.], [0., 0., 0.]])
10 #another_example로 test
11 compute_mean_embedding(another_example)

```

```

1 embedding_size = 20
2
3 model = keras.models.Sequential([
4     text_vectorization,
5     keras.layers.Embedding(input_dim=n_tokens,
6                             output_dim=embedding_size,
7                             mask_zero=True), #<pad> token을 zero vector로 변환
8     keras.layers.Lambda(compute_mean_embedding),
9     keras.layers.Dense(100, activation="relu"),
10    keras.layers.Dense(1, activation="sigmoid"),
11 ])

```

과 제 할 당	14.1 고성호 14.2 서가을 14.3 이아현 14.4 안세윤 14.5 허주희 14.6 허주희 14.7 하정현 14.8 하정현 14.9 이문기 14.10 박제윤 14.11 권기호
특 이 사 항	8기 서가을 백신접종 및 후유증으로 인해 미참석
비 고	10.15, 10.22, 10.29 휴동