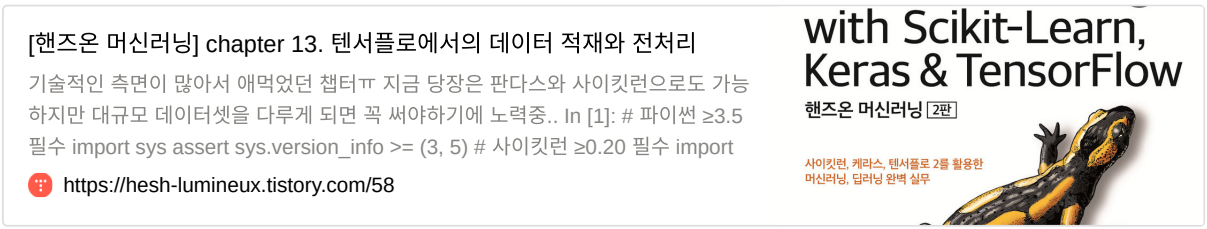
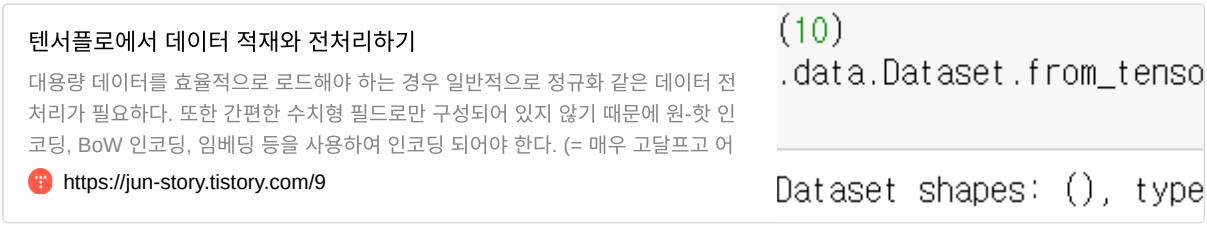
Ch13_텐서플로에서 데이터 적재와 전처리하기 (1)





- 텐서플로는 데이터 API로 대규모 데이터셋을 쉽게 처리할 수 있다.
- `TFRecord`는 프로토콜 버퍼(protocol buffer)를 담은 유연하고 효율적인 이진 포맷이다.
- **TF 변환(tf.Transform)** : (실행 속도를 높이기 위해) 훈련 전에 전체 훈련 세트에 대해 실행하는 전처리 함수를 작성할 수 있다.
- **TF 데이터셋 (TFDS)** : 각종 데이터셋을 다운로드할 수 있는 편리한 함수를 제공한다.
  - 이미지넷과 같은 대용량 데이터셋도 포함된다.
  - 데이터 API로 조작할 수 있는 편리한 데이터셋 객체도 제공한다.

▼ 13.1 데이터 API

• **데이터셋(dataset)**은 **연속된 데이터 샘플**을 나타낸다.

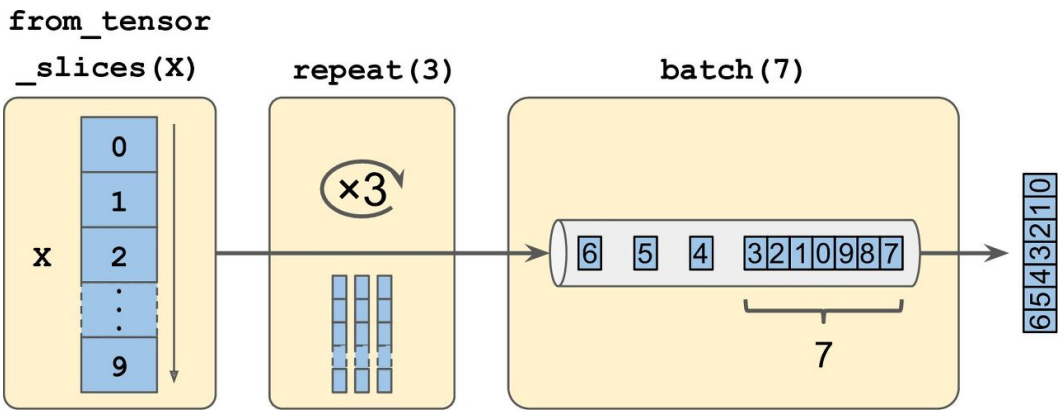
• `from_tensor_slices()` : 텐서를 받아 X의 각 원소가 아이템(item)으로 표현되는 데이터셋(tf.data.Dataset)을 만든다.

```
import tensorflow as tf

X = tf.range(10)

dataset = tf.data.Dataset.from_tensor_slices(X)
dataset
# <TensorSliceDataset shapes: (), types: tf.int32>

for item in dataset:
    print(item)

# tf.Tensor(0, shape=(), dtype=int32)
# tf.Tensor(1, shape=(), dtype=int32)
# tf.Tensor(2, shape=(), dtype=int32)
# tf.Tensor(3, shape=(), dtype=int32)
# tf.Tensor(4, shape=(), dtype=int32)
# tf.Tensor(5, shape=(), dtype=int32)
# tf.Tensor(6, shape=(), dtype=int32)
# tf.Tensor(7, shape=(), dtype=int32)
# tf.Tensor(8, shape=(), dtype=int32)
# tf.Tensor(9, shape=(), dtype=int32)
```

▼ 13.1.1 연쇄 변환

데이터셋이 준비되면 변환 메서드를 호출하여 여러 종류의 변환을 수행할 수 있다. 각 메서드는 새로운 데이터셋을 반환하므로 변환 데이터를 연결할 수 있다.

```
dataset =  dataset.repeat(3).batch(7)
for item in dataset:
    print(item)

# tf.Tensor([0 1 2 3 4 5 6], shape=(7,), dtype=int32)
# tf.Tensor([7 8 9 0 1 2 3], shape=(7,), dtype=int32)
# tf.Tensor([4 5 6 7 8 9 0], shape=(7,), dtype=int32)
# tf.Tensor([1 2 3 4 5 6 7], shape=(7,), dtype=int32)
# tf.Tensor([8 9], shape=(2,), dtype=int32)
```



→ 원본 데이터셋(아이템 10개)를 세 차례 반복하고 (repeat 3), 생성된 총 30개의 값을 7개의 배치로 나눠 각 텐서에 저장한다. (batch 7)

- `repeat()` : 원본 데이터셋의 아이템을 세 차례 반복하는 새로운 데이터셋을 반환한다.

새로운 데이터셋에서 `batch()`를 호출하면 다시 새로운 데이터셋이 만들어진다.

- `batch()` 메서드를 `drop_remainder=True`로 호출하면 길이가 모자란 마지막 배치를 버리고 모든 배치를 동일한 크기로 맞춘다.

- **변환 적용하는 메서드**

- `map()` 메서드를 호출하여 아이템을 변환할 수도 있다.

```
#모든 아이템에 *2를 하여 새로운 데이터셋 만들기
dataset = dataset.map(lambda x: x* 2)
```

- `filter()` 메서드를 사용하여 데이터셋을 필터링 한다.

```
dataset = dataset.filter(lambda x: x < 10)
```

- `take()` : 보고 싶은 몇 개의 아이템만을 보여주는 메서드

```
for item in dataset.take(3):
    print(item)

# tf.Tensor([ 0  8 16 24 32 40 48], shape=(7,), dtype=int32)
# tf.Tensor([56 64 72  0  8 16 24], shape=(7,), dtype=int32)
# tf.Tensor([32 40 48 56 64 72  0], shape=(7,), dtype=int32)
```

- `apply()` : 데이터셋 전체에 변환을 적용하는 메서드

▼ 13.1.2 데이터 셔플링

경사 하강법은 훈련 세트에 있는 샘플이 독립적이고 동일한 분포일 때 최고의 성능을 발휘한다.

텐서로 이루어진 훈련 샘플을 효율적으로 섞는 방법은 `shuffle()`을 이용하는 것 이다.

```
dataset = tf.data.Dataset.range(10).repeat(3) # 0에서 9까지 세 번 반복
dataset = dataset.shuffle(buffer_size=5, seed=42).batch(7)
```

```
for item in dataset:
    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()` 는 원본 데이터셋의 처음 아이템을 `buffer_size` 개수만큼 추출하여 버퍼에 채운다. 그 다음 새로운 아이템이 요청되면 이 버퍼에서 랜덤하게 하나를 꺼내 반환한다. 그리고 원본 데이터셋에서 새로운 아이템을 추출하여 비워진 버퍼를 채운다. 이 과정은 원본 데이터셋의 모든 아이템이 사용될 때까지 반복된다.

메모리 용량보다 큰 대규모 데이터셋은 버퍼가 데이터셋에 비해 작기 때문에 해결책으로 원본 데이터 자체를 섞는게 좋다.

`buffer_size`의 크기는 데이터셋의 크기와 비슷할 수록 좋다. 그 이유는 버퍼의 크기가 작으면 원본 데이터셋에서 뒤쪽에 있는 아이템이 새로 만들어진 데이터셋에서도 뒤에 등장할 가능성이 높기 때문이다.

- **여러 파일에서 한 줄씩 번갈아 읽기**

```
#캘리포니아 주택 데이터셋 적재하기
from sklearn.datasets import fetch_california_housing
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

housing = fetch_california_housing()
X_train_full, X_test, y_train_full, y_test = train_test_split(
    housing.data, housing.target.reshape(-1, 1), random_state=42)
X_train, X_valid, y_train, y_valid = train_test_split(
    X_train_full, y_train_full, random_state=42)

scaler = StandardScaler()
scaler.fit(X_train)
X_mean = scaler.mean_
X_std = scaler.scale_

#주택 데이터셋을 20개의 CSV파일로 나누기
def save_to_multiple_csv_files(data, name_prefix, header=None, n_parts=10):
    housing_dir = os.path.join("images", "data")
    os.makedirs(housing_dir, exist_ok=True)
    path_format = os.path.join(housing_dir, "my_{}_{:02d}.csv")

    filepaths = []
    m = len(data)
    for file_idx, row_indices in enumerate(np.array_split(np.arange(m), n_parts)):
        part_csv = path_format.format(name_prefix, file_idx)
        filepaths.append(part_csv)
        with open(part_csv, "wt", encoding="utf-8") as f:
            if header is not None:
                f.write(header)
```

```
                f.write("\n")
            for row_idx in row_indices:
                f.write(",".join([repr(col) for col in data[row_idx]]))
                f.write("\n")
    return filepaths

train_data = np.c_[X_train, y_train]
valid_data = np.c_[X_valid, y_valid]
test_data = np.c_[X_test, y_test]
header_cols = housing.feature_names + ["MedianHouseValue"]
header = ",".join(header_cols)

train_filepaths = save_to_multiple_csv_files(train_data, "train", header, n_parts=20)
valid_filepaths = save_to_multiple_csv_files(valid_data, "valid", header, n_parts=10)
test_filepaths = save_to_multiple_csv_files(test_data, "test", header, n_parts=10)

train_filepaths

# 입력 파이프라인 만들기
# 기본적으로 shuffle이 설정 / shuffle = False 가능
filepath_dataset = tf.data.Dataset.list_files(train_filepaths, seed=42)
```

기본적으로 `list_files()` 함수는 파일 경로를 섞은 데이터셋을 반환한다. 그 다음 `interleave()` 메서드를 호출하여 한 번에 다섯 개의 파일을 한 줄씩 번갈아 읽는다.

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

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

▼ 13.1.3 데이터 전처리

```
n_inputs = 8

def preprocess(line):
    defs = [0.] * n_inputs + [tf.constant([], dtype=tf.float32)]
    fields = tf.io.decode_csv(line, record_defaults = defs)
    x = tf.stack(fields[:-1])
    y = tf.stack(fields[-1:])
    return (x - X_mean) / X_std, y
```

```
preprocess(b'4.2083,44.0,5.3232,0.9171,846.0,2.3370,37.47,-122.2,2.782')
```

- 먼저 이 코드는 훈련 세트에 있는 각 특성의 평균과 표준편차를 미리 계산했다고 가정합니다. X_mean와 X_std는 특성마다 1개씩 8개의 실수를 가진 1D 텐서(또는 넘파이 배열)입니다.
- preprocess() 함수는 CSV 한 라인을 받아 파싱합니다. 이를 위해 tf.io.decode_csv() 함수를 사용합니다. 이 함수는 두 개의 매개변수를 받습니다. 첫 번째는 파싱할 라인과 두 번째는 CSV 파일의 각 열에 대한 기본값을 담은 배열입니다. 이 배열은 텐서플로에게 각 열의 기본값뿐만 아니라 열 개수과 데이터 타입도 알려줍니다. 이 예에서 모든 특성 열이 실수이고 누락된 값의 기본값은 0으로 지정했습니다. 마지막 열(타깃)에 tf.float32 타입의 빈 배열을 제공합니다. 이 배열은 텐서플로에게 이 열은 실수이지만 기본값이 없다고 알려줍니다. 따라서 이 열에서 누락된 값이 발견되면 예외가 발생할 것입니다.
- decode_csv() 함수는 (열마다 한 개씩) 스칼라 텐서의 리스트를 반환합니다. 1D 텐서 배열을 반환해야 하므로 마지막 열(타깃)을 제외하고 모든 텐서에 대해 tf.stack() 함수를 호출합니다. 이 함수는 모든 텐서를 쌓아 1D 배열을 만듭니다. 그다음 타깃값에도 동일하게 적용합니다(이렇게 하면 스칼라 텐서가 아니라 하나의 값을 가진 1D 텐서가 됩니다).
- 마지막으로 입력 특성에서 평균을 빼고 표준편차로 나누어 스케일을 조정합니다. 그다음 스케일 조정된 특성과 타깃을 담은 튜플을 반환합니다.

▼ 13.1.4 데이터 적재와 전처리 합치기

CSV 파일에서 캘리포니아 주택 데이터셋을 효율적으로 적재하고, 전처리, 셔플링, 반복, 배치를 적용한 데이터셋을 만들어 반환한다.

```
def csv_reader_dataset(filepaths, repeat=1, n_readers=5,
                       n_read_threads=None, shuffle_buffer_size=10000,
                       n_parse_threads=5, batch_size=32):
    dataset = tf.data.Dataset.list_files(filepaths).repeat(repeat)
    dataset = dataset.interleave(
        lambda filepath: tf.data.TextLineDataset(filepath).skip(1),
        cycle_length=n_readers, num_parallel_calls=n_read_threads)
    dataset = dataset.shuffle(shuffle_buffer_size)
    dataset = dataset.map(preprocess, num_parallel_calls=n_parse_threads)
    dataset = dataset.batch(batch_size)
    return dataset.prefetch(1)
```

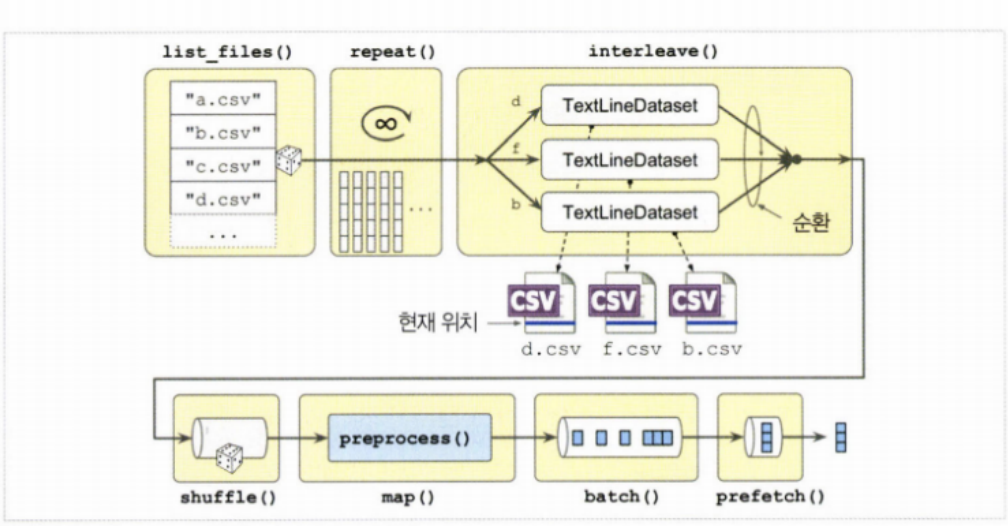


그림 13-2 여러 CSV 파일에서 데이터를 적재하고 전처리하기

```
tf.random.set_seed(42)

train_set = csv_reader_dataset(train_filepaths, batch_size=3)
for X_batch, y_batch in train_set.take(2):
    print("X =", X_batch)
    print("y =", y_batch)
    print()

>>>
X = tf.Tensor(
[[ 0.5804519  -0.20762321  0.05616303 -0.15191229  0.01343246  0.00604472
   1.2525111  -1.3671792 ]
 [ 5.818099    1.8491895   1.1784915   0.28173092 -1.2496178  -0.3571987
   0.7231292  -1.0023477 ]
 [-0.9253566   0.5834586  -0.7807257  -0.28213993 -0.36530012  0.27389365
  -0.76194876  0.72684526]], shape=(3, 8), dtype=float32)

y = tf.Tensor(
[[1.752]
 [1.313]
 [1.535]], shape=(3, 1), dtype=float32)

X = tf.Tensor(
[[-0.8324941   0.6625668  -0.20741376 -0.18699841 -0.14536144  0.09635526
   0.9807942  -0.67250353]
 [-0.62183803  0.5834586  -0.19862501 -0.3500319  -1.1437552  -0.3363751
   1.107282   -0.8674123 ]
 [ 0.8683102   0.02970133  0.3427381  -0.29872298  0.7124906   0.28026953
  -0.72915536  0.86178064]], shape=(3, 8), dtype=float32)

y = tf.Tensor(
[[0.919]
 [1.028]
 [2.182]], shape=(3, 1), dtype=float32)
```

▼ 13.1.5 프리패치

마지막에 `prefetch(1)`를 호출하면 데이터셋은 항상 한 배치가 미리 준비되도록 최선을 다한다. 다른 말로 하면 알고리즘이 한 배치로 작업을 하는 동안 이 데이터셋이 동시에 다음 배치를 준비한다.

멀티스레드로 데이터를 적재하고 전처리하면 CPU의 멀티 코어를 활용하여 아마도 GPU에서 훈련 스텝을 수행하는 것보다 짧은 시간안에 한 배치 데이터를 준비할 수 있다.

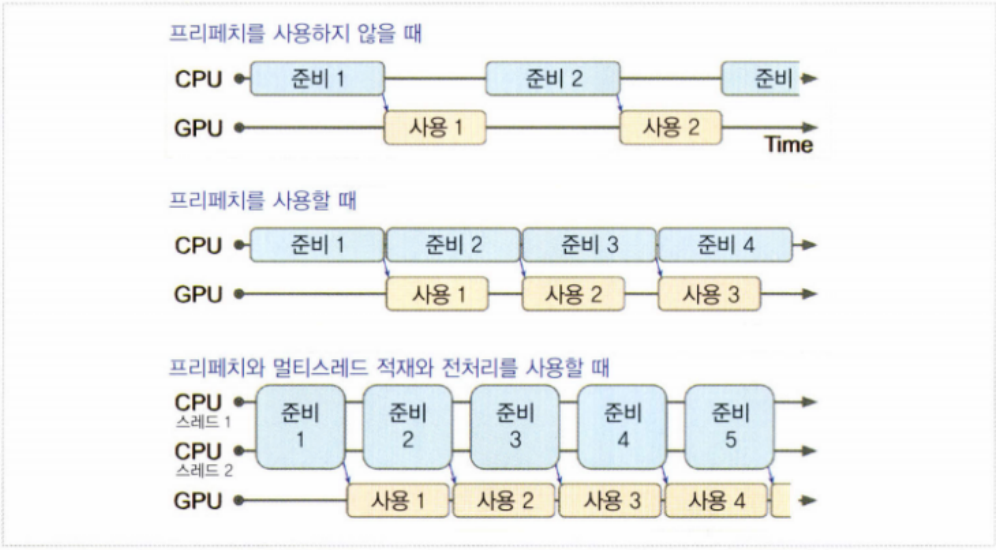


그림 13-3 프리페치로 CPU와 GPU를 동시에 사용합니다. GPU가 한 배치를 처리할 때 CPU가 그다음 배치를 준비합니다.

▼ 13.1.6 tf.keras와 데이터셋 사용하기

▼

`csv_reader_dataset()` 함수로 훈련 세트로 사용할 데이터셋을 만들 수 있다.

`tf.keras`에서 반복을 처리하므로 반복을 지정할 필요가 없다.

```
train_set = csv_reader_dataset(train_filepaths, repeat=None)
valid_set = csv_reader_dataset(valid_filepaths)
test_set = csv_reader_dataset(test_filepaths)
```

이제 케라스 모델을 만들고 이 데이터셋으로 훈련할 수 있다.

`fit()`메서드에 X_train, y_train, X_valid, y_valid 대신 훈련 데이터셋과 검증 데이터셋을 전달하기만 하면 된다.

```
model = keras.models.Sequential([
    keras.layers.Dense(30, activation="relu", input_shape=X_train.shape[1:]),
```

```
    keras.layers.Dense(1),
])

model.compile(loss="mse", optimizer=keras.optimizers.SGD(lr=1e-3))
```

```
batch_size = 32
model.fit(train_set, steps_per_epoch=len(X_train) // batch_size, epochs=10,
          validation_data=valid_set)
```

비슷하게 `evaluate()`와 `predict()` 메서드에 데이터셋을 전달할 수 있다.

```
model.evaluate(test_set)
new_set = test_set.take(3).map(lambda X, y : X)
model.predict(new_set)
```

자신만의 훈련 반복을 만들고 싶다면 그냥 훈련 세트를 반복하면 된다.

```
#자신만의 훈련 반복
optimizer = keras.optimizers.Nadam(lr=0.01)
loss_fn = keras.losses.mean_squared_error

n_epochs = 5
batch_size = 32
n_steps_per_epoch = len(X_train) // batch_size
total_steps = n_epochs * n_steps_per_epoch
global_step = 0
for X_batch, y_batch in train_set.take(total_steps):
    global_step += 1
    print("\rGlobal step {}/{}".format(global_step, total_steps), end="")
    with tf.GradientTape() as tape:
        y_pred = model(X_batch)
        main_loss = tf.reduce_mean(loss_fn(y_batch, y_pred))
        loss = tf.add_n([main_loss] + model.losses)
    gradients = tape.gradient(loss, model.trainable_variables)
    optimizer.apply_gradients(zip(gradients, model.trainable_variables))
```

- 전체 훈련 반복을 수행하는 텐서플로 함수

```
#전체 훈련 반복을 수행하는 텐서플로 함수
optimizer = keras.optimizers.Nadam(lr=0.01)
loss_fn = keras.losses.mean_squared_error

@tf.function
def train(model, n_epochs, batch_size=32,
          n_readers=5, n_read_threads=5, shuffle_buffer_size=10000, n_parse_threads=5):
    train_set = csv_reader_dataset(train_filepaths, repeat=n_epochs, n_readers=n_readers,
                       n_read_threads=n_read_threads, shuffle_buffer_size=shuffle_buffer_size,
                       n_parse_threads=n_parse_threads, batch_size=batch_size)
    for X_batch, y_batch in train_set:
        with tf.GradientTape() as tape:
```

```
            y_pred = model(X_batch)
            main_loss = tf.reduce_mean(loss_fn(y_batch, y_pred))
            loss = tf.add_n([main_loss] + model.losses)
        gradients = tape.gradient(loss, model.trainable_variables)
        optimizer.apply_gradients(zip(gradients, model.trainable_variables))

train(model, 5)
```

▼ 13.2 TFRecord 포맷

TFRecord

대용량 데이터를 저장하고 효율적으로 읽기 위해 텐서플로가 선호하는 포맷으로, 크기가 다른 연속된 이진 레코드를 저장하는 단순한 이진 포맷이다.

`tf.io.TFecordWriter` 클래스를 사용해 TRecord를 손쉽게 만들 수 있다.

```
with tf.io.TFRecordWriter("my_data.tfrecord") as f:
    f.write(b"데이터를 작성할 때는 바이트 객체로 작성하자")
    f.write(b"그리고 이건 두번째 레코드가 된다")
```

`f.data.TFRecordDataset()`을 사용하여 하나 이상의 TFRecord를 읽을 수 있다.

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

TIP 기본적으로 TFRecordDataset는 파일을 하나씩 차례로 읽습니다. num_parallel_reads을 지정하여 여러 파일에서 레코드를 번갈아 읽을 수 있습니다. 또는 앞서 CSV 파일에 적용했던 것처럼 list_files()와 interleave()을 사용하여 동일한 결과를 얻을 수 있습니다.

압축된 TFRecord 파일을 읽으려면 압축 형식을 지정해주면 된다

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

▼ 13.2.1 압축된 TFRecord 파일

네트워크를 통해 읽어야 하는 경우, TFRecord을 압축해야 할 때 `option` 객체를 이용한다.

```
#options 매개변수를 사용하여 압축된 TFRecord 파일 만들기
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")
```

만약 이렇게 압축된 TFRecord 파일을 읽으려면 압축 형식을 지정해주면 된다.

```
#압축된 TFRecord 파일을 읽으려면 압축 형식을 지정해야함
dataset = tf.data.TFRecordDataset(["my_compressed.tfrecord"],
                                  compression_type="GZIP")
for item in dataset:
    print(item)
```

▼ 13.2.2 프로토콜 버퍼 개요

각 레코드는 어떤 이진 포맷도 사용할 수 있찌만 일바적으로 TFRecord는 직렬화된 프로토콜 버퍼(protocol buffer)를 담고 있다.

```
syntax = "proto3"
message Person{
    string name = 1;
    int32 id = 2;
    repeated string email = 3;
}
```

```
from person_pb2 import Person # 생성된 클래스를 잉포트합니다.
))) person = Person (name="Al" , id=123, email=[ .. a@b .com .. ]) # Person 객체를 만듭니다.
))) print(person) # Person을 출력합니다.
name: "Al "
id: 123
email: .. a@b.com ..
))) person .name # 필드를 읽습니다.
1'Al "
))) person .name = "Alice" # 필드를 수정합니다
))) person .email[0] # 반복 필드는 배열처럼 참조할 수 있습니다.
"a@b. com"
))) person .email. append ( .. c@d .com .. ) # 이메일 주소를 추가합니다.
))) s = person .SerializeToString() # 바이트 운자열로 객체를 직렬화합니다.
))) s
b ' \ n\x0SAlice\x10{\x1a\x07a@b .com\x1a\x07c@d.com'
))) person2 = Person () # 새로운 Person 객체를 만듭니다.
))) person2 .ParseFromString(s) # 바이트 문자열을 파싱합니다 (27 바이트 길이입니다) .
27
```

```
))) person == person2 # 두 객체는 동일합니다.
True
```

`protoc`로 생성된 `Person` 클래스를 임포트하고 객체를 만들어 출력해보고 필드의 값을 읽고 써보았다. 그다음 `SerializeToString()` 메서드를 사용해 직렬화. 이 문자열을 저장하거나 네트워크를 통해 전달할 수 있다. 이진 데이터를 읽거나 수신하면 `ParseFromString()` 메서드를 사용해 파싱할 수 있다. 이를 통해 직렬화한 객체의 복사본을 얻을 수 있다.

▼ 13.2.3 텐서플로 프로토콜 버퍼

TFRecord 파일에서 사용하는 전형적인 주요 프로토콜 버퍼는 데이터셋에 있는 하나의 샘플을 표현하는 Example 프로토콜 버퍼이다. 이 프로토콜 버퍼는 이름을 가진 특성의 리스트를 가지고 있다.

각 특성은 바이트 스트링의 리스트나 실수의 리스트, 정수의 리스트 중 하나이다.

```
syntax = "proto3";

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

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

Feature는 바이트, 실수, 정수 리스트중 하나를 가지고 있다.
또한 Features는 string과 대응되는 Feature가 있는 딕셔너리를 가진다.

마지막으로 Example은 Features 객체 하나를 가진다. 따라서 아래와 같은 코드를 제작할 수 있다.

- Person과 동일하게 표현한 tf.train.Example 객체를 만들고 TFRecord 파일에 저장하는 방법

```
from tensorflow.train import BytesList, FloatList, Int64List
from tensorflow.train import Feature, Features, Example

person_example = Example(
    # Example은 features라는 이름의 Features객체를 가지고 있으며 이는 딕셔너리 리스트이다.
    features=Features(
```

```
        # 또다시 Features는 feature라는 이름의 Feature객체를 가지고 이는 딕셔너리파일 이다.
        feature={
            # 그리고 Feature는 정수, 실수, 문자열 리스트중 하나를 가지고 있다.
            "name": Feature(bytes_list=BytesList(value=[b"Alice"])),
            "id": Feature(int64_list=Int64List(value=[123])),
            "emails": Feature(bytes_list=BytesList(value=[b"a@b.com", b"c@d.com"]))
        }))

# 이제 Example 프로토콜 버퍼가 생겼으므로 SerializeToString()을 통하여 직렬화 하고 TFRecord에 저장한다.
with tf.io.TFRecordWriter("my_contacts.tfrecord") as f:
    f.write(person_example.SerializeToString())
```

Example 프로토콜 버퍼를 만들었으면 `SerializeToStrain()` 메서드를 호출하여 직렬화하고 결과 데이터를 TFRecord 파일에 저장할 수 있다.

```
with tf.io.TFRecordWriter( ''my_contacts. tfrecord" ) as f:
f.write(person_examp1e .Seria1izeToString())
```

▼ 13.2.3 Example 프로토콜 버퍼를 읽고 파싱하기

직렬화된 Example 프로토콜 버퍼를 읽기 위해 `tf.data.TFRecordDataset`을 다시 한번 사용하고 `tf.io.parse_single_example()`을 사용하여 각 Example을 파싱하자.

`parse_single_example()`에는 두가지 파라미터가 필요한데 직렬화된 데이터와 각 특성에 대한 설명이다. 이 설명은 각 특성 이름에 대해 특성의 크기, 타입, 기본값을 표현한 `tf.io.FixedLenFeature`이나 특성 타입만 표현된 `tf.io.VarLenFeature`에 매핑한 딕셔너리 이다.

다음은 설명 닥셔너리를 정의하고 TFRecordDataset을 순회하면서 데이터셋에 포함된 직렬화된 Example 프로토콜 버퍼를 파싱하는 코드다.

```
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"]):
    # 결과는 딕셔너리 객체로 나온다.
    parsed_example = tf.io.parse_single_example(serialized_example,
                                                feature_description)

parsed_example
```

`FixedLenFeature(고정 길이 특성)`는 보통의 텐서로 파싱되어 나오지만 `VarLenFeature(가변 길이 특성)`는 희소 텐서로 파싱된다.

`tf.sparse.to_dense()`를 통해 밀집 텐서로 변환할 수 있지만 여기에서는 희소 텐서의 값을 바로 참조하는게 편하다.

```
tf.spares.to_dense(pare
```

BtyesList는 직렬화된 객체를 포함하여 모든 이진 데이터를 포함할 수 있다.

특히 이미지 데이터를 `tf.io.encode_jpeg()`를 사용하여 jpeg 포맷 이미지를 인코딩하고, 이 이진 데이터를 `ByteList`에 넣어서 나중에 Example을 파싱하고 ,다시 여기서 데이터를 파싱한 뒤, 이 데이터를 `tf.io.decode_jpeg()`에 넘겨서 원본 이미지를 얻을 수 있다.

```
from sklearn.datasets import load_sample_images

img = load_sample_images()["images"][0]
plt.imshow(img)
plt.axis("off")
plt.title("Original Image")
plt.show()
```

```
data = tf.io.encode_jpeg(img)
example_with_image = Example(features=Features(feature={
    "image": Feature(bytes_list=BytesList(value=[data.numpy()]))}))
serialized_example = example_with_image.SerializeToString()
# then save to TFRecord
```

```
feature_description = { "image": tf.io.VarLenFeature(tf.string) }
example_with_image = tf.io.parse_single_example(serialized_example, feature_description)
decoded_img = tf.io.decode_jpeg(example_with_image["image"].values[0])
```

또는 BMP, GIF, JPEG, PNG 이미지를 디코딩할 수 있는 `tf.io.decode_image()` 를 사용한다.

또한 `tf.io.serialize_tensor()` 를 사용하여 어떤 텐서라도 직렬화하고, 결과 바이트 스트링을 `BytesList` 특성 에 넣어 저장할 수 있다.

나중에 이 TFRecord를 파싱할때는 `tf.io.parse tensor(`)를 사용하여 이 데이터 파싱한다.

`tf.io.parse single ample ()` 로 하나씩 파싱하는 대신 `tf.io.parse ample()` 를 사용하여 배치 단위로 파싱할 수 있다.

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

parsed_examples
```

▼ 13.2.3 SequenceExample 프로토콜 버퍼를 사용해 리스트의 리스트(순차 데이터) 다루기

- SequenceExample 프로토콜 버퍼의 정의

```
syntax = "proto3";

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

`SequenceExample`은 문맥 데이터를 위한 하나의 `Features 객체`와 이름이 있는 한 개 이상의 `FeatureList`를 가진 `FeatureLists 객체`를 포함한다.

특성 리스트가 가변 길이의 시퀀스를 담고 있다면 `tf.RaggedTensor.from_sparse()`를 사용해 래그드 텐서로 바꿀 수 있다.

```
from tensorflow.train import FeatureList, FeatureLists, SequenceExample

context = Features(feature={
    "author_id": Feature(int64_list=Int64List(value=[123])),
    "title": Feature(bytes_list=BytesList(value=[b"A", b"desert", b"place", b"."])),
    "pub_date": Feature(int64_list=Int64List(value=[1623, 12, 25]))
})

content = [["When", "shall", "we", "three", "meet", "again", "?"],
           ["In", "thunder", ",", "lightning", ",", "or", "in", "rain", "?"]]
comments = [["When", "the", "hurlyburly", "'s", "done", "."],
            ["When", "the", "battle", "'s", "lost", "and", "won", "."]]

def words_to_feature(words):
    return Feature(bytes_list=BytesList(value=[word.encode("utf-8")
                                               for word in words]))

content_features = [words_to_feature(sentence) for sentence in content]
comments_features = [words_to_feature(comment) for comment in comments]

sequence_example = SequenceExample(
    context=context,
    feature_lists=FeatureLists(feature_list={
        "content": FeatureList(feature=content_features),
```

```
        "comments": FeatureList(feature=comments_features)
    }))

sequence_example
```

```
serialized_sequence_example = sequence_example.SerializeToString()
```

```
context_feature_descriptions = {
    "author_id": tf.io.FixedLenFeature([], tf.int64, default_value=0),
    "title": tf.io.VarLenFeature(tf.string),
    "pub_date": tf.io.FixedLenFeature([3], tf.int64, default_value=[0, 0, 0]),
}
sequence_feature_descriptions = {
    "content": tf.io.VarLenFeature(tf.string),
    "comments": tf.io.VarLenFeature(tf.string),
}
parsed_context, parsed_feature_lists = tf.io.parse_single_sequence_example(
    serialized_sequence_example, context_feature_descriptions,
    sequence_feature_descriptions)

parsed_context

print(tf.RaggedTensor.from_sparse(parsed_feature_lists["content"]))
```

▼ 13.3 입력 특성 전처리

신경망을 위해 데이터를 준비하려면 일반적으로 모든 특성을 수치 특성으로 변환하고 정규화 해야 한다.

특히 데이터에 범주형 특성이나 텍스트 특성이 있다면 숫자로 바꾸어야 한다.

- Lambda 층을 사용해 표준화를 수행하는 층 구현

```
#Lambda 층을 사용해 표준화를 수행하는 층을 구현
mean = np.maen(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))
    ...
])
```

- 완전한 사용자 정의 층

```
#전역변수를 다루기보다 StandardScaler처럼 완전한 사용자 정의 층 정의
class Standardization(keras.layers.Layer):
    def adapt(self, data_sample):
```

```
        self.means_ = np.mean(data_sample, aixs=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())
```

Standardization 층을 모델에 추가하기 전에 데이터 샘플과 함께 `adapt()`메서드를 호출해야 한다. 이렇게 해야 각 특성에 대해 적절한 평균과 표준편차를 사용할 수 있다.

```
#Standardization층을 모델에 추가하기 전에 데이터 샘플과 함께 adapt()메서드를 호출
std_layer = Standardization()
std_layer.adapt(data_sample)
```

▼ 13.3.1 원-핫 벡터를 사용해 범주형 특성 인코딩하기

범주 개수가 매우 작을때 원-핫 인코딩을 사용할 수 있다.

이를 위해 먼저 룩업 테이블을 사용해 각 범주를 인덱스로 매핑한다.

```
vocab = ['<1H OCEAN', 'INLAND', 'ISLAND', 'NEAR BAY', 'NEAR OCEAN']
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)
```

- 먼저 **어휘 사전**vocabulary을 정의합니다. 이는 가능한 모든 범주의 리스트입니다.
- 그다음 범주에 해당하는 인덱스(0에서 4까지)의 텐서를 만듭니다.
- 다음으로 범주 리스트와 해당 인덱스를 전달하여 룩업 테이블을 위해 초기화 객체를 만듭니다. 이 예에서는 이미 이 데이터를 가지고 있으므로 `KeyValueTensorInitializer`를 사용합니다. 하지만 범주가 텍스트 파일에 (라인당 하나의 범주로) 나열되어 있다면 `TextFileInitializer`를 사용합니다.
- 마지막 두 라인에서 초기화 객체와 **oov**out-of-vocabulary 버킷bucket을 지정하여 룩업 테이블을 만듭니다. 어휘 사전에 없는 범주를 찾으면 룩업 테이블이 계산한 이 범주의 해시값을 이용하여 oov 버킷 중 하나에 할당합니다. 인덱스는 알려진 범주 다음부터 시작합니다. 따라서 이 예제에서 두 개의 oov 버킷의 인덱스는 5와 6입니다.

oov버킷을 사용하는 이유는 데이터셋이 크거나 범주가 자주 바뀐다면 전체 범주 리스트를 구하는 것이 어려울 수 있다. 따라서 전체 훈련 세트가 아닌 샘플 데이터를 기반으로 어휘 사전을 정의하고 샘플 데이터에 없는 다른 범주를 oov버킷에 추가하는 것이다.

훈련 도중 발견되는 알려지지 않은 범주가 많을수록 더많은 oov버킷을 사용해야 한다. oov버킷이 충분하지 않으면 충돌이 발생할 수 있다.

```
categories = tf.constant(['NEAR BAY', 'DESERT', 'INLAND', 'INLAND'])
cat_indices = table.lookup(categories)
cat_indices
```

```
cat_one_hot = tf.one_hot(cat_indices, depth =len(vocab) + num_oov_buckets)
cat_one_hot
```

가능한 범주가 몇 개되지 않을 때는 괜찮지만 어휘 사전이 크면 임베딩(embedding)을 사용하여 인코딩하는 것이 효율적이다.

경험적으로 보았을 때 범주 개수가 10 이하이면 일반적으로 원-핫 인코딩을 사용한다. 범주 개수가 50개 이상이면 임베딩이 선호되는 편이다.

▼ 13.3.2 임베딩을 사용해 범주형 특성 인코딩하기

> 임베딩

- 임베딩은 범주를 표현하는 훈련 가능한 밀집 벡터이다.
- 처음엔 임베딩이 랜덤하게 초기화되어 있다.
- 처음에는 랜덤 벡터로 표현되고, 차원 수는 수정 가능한 하이퍼파라미터이다.
- 임베딩을 훈련할 수 있기 때문에 훈련 도중에 점차 향상된다.
- 비슷한 범주들은 경사 하강법이 더 가깝게 만들 것이다.
- 범주가 유용하게 표현되도록 임베딩이 훈련되는 경향을 표현 학습(Representation Learning)이라고 부른다.

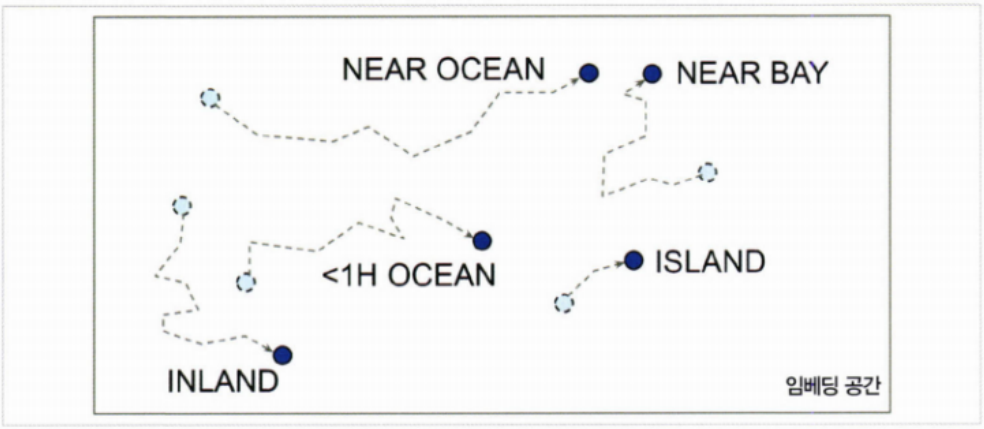


그림 13-4 임베딩이 훈련하는 동안 점차 향상됩니다.

- 임베딩 직접 구현하기

```
# 임베딩 행렬(Embedding Matrix)을 만들어 초기화
embedding_dim = 2
embed_init = tf.random.uniform([len(vocab) + num_oov_buckets, embedding_dim])
embedding_matrix = tf.Variable(embed_init)
embedding_matrix
```

```
#동일한 범주 특성을 인코딩
categorise = tf.constant(["NEAR BAY", "DESERT", "INLAND", "INLAND"])
cat_indices = table.lookup(categories)
cat_indices
```

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

```
tf.nn.embedding_lookup(embedding_matrix, cat_indices)
```

케라스는 기본적으로 학습 가능한 임베딩 행렬을 처리해주는 `keras.layers.Embedding 층`을 제공한다. 이 층이 생성될 때 임베딩 행렬을 랜덤하게 초기화하고 어떤 범주 인덱스로 호출될 때 임베딩 행렬에 있는 그 인덱스의 행을 반환한다.

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

```
#모두 연결하면 범주형 특성을 처리하고 각 범주마다 임베딩을 학습하는 케라스 모델
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])
```

이 모델은 두 개의 입력을 받습니다. 샘플마다 8개의 특성을 담은 입력과 (샘플마다 하나의 범주형 특성을 담은) 하나의 범주형 입력입니다. Lambda 층을 사용해 범주의 인덱스를 찾은 다음 임베딩에서 이 인덱스를 찾습니다. 그다음 이 임베딩과 일반 입력을 연결하여 신경망에 주입할 인코드된 입력을 만듭니다. 여기서부터는 어떤 신경망도 추가할 수 있지만 간단하게 완전 연결 층을 하나 추가하여 케라스 모델을 만듭니다.

▼ 13.3.3 케라스 전처리 층

- keras.layers.Normalization층 : 특성을 표준화를 수행
- TextVectorization 층 : 입력에 있는 각 단어를 어휘 사전에 있는 인덱서를 인코등한다.
- 두 경우 모두 층을 만들고 샘플 데이터로 adapt()메서드를 호출한 다음 일반적인 층처럼 모델에 사용할 수 있다.
- keras.layers.Discretization층 : 연속적인 데이터를 몇 개의 구간으로 나누고, 각 구간을 원-핫 벡터로 인코딩한다.
- PreprocessingStage클래스를 사용해 여러 전처리 층을 연결할 수 있다.
- 예시로 입력을 정규화하고, 그 다음 이산화(Discretization)하는 전처리 파이프라인을 만든다. 이 파이프라인을 샘플 데이터에 적응시킨 다음 일반적인 층처럼 모델에 사용할 수 있다.

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

- TextVectorization층에 있는 옵션들 :
- BOW(bag of word) : 단어의 순서를 무시하고, 단어 인데스 대신 단어 카운트 벡터를 출력하는 옵션을 말한다.
- 흔하게 등장하는 단어는 흥미로운 정보가 없다고 판단한다.
- 따라서 단어 카운트는 자주 등장하는 단어의 중요도를 줄이는 방향으로 정규화되어야 한다.
- 자주 사용하는 정규화하는 방법으로는 TF-IDF(Term Frequency-Inverse Document Frequency)
- 전체 샘플 수를 단어가 등장하는 훈련 샘플 개수로 나눈 로그를 계산한 후 단어 카운트와 곱하는 것

▼ 13.4 TF 변환

- 전처리는 계산 비용이 크기 때문에 훈련과 동시에 수행하는 것보다 사전에 처리하면 속도를 크게 높일 수 있다.

- 데이터가 아주 크면 아파치 빔이나 스파크 같은 도구가 도움 이된다.
- 이런 도구로 대규모 데이터에서 효율적인 데이터 처리 파이프라인을 수행할 수 있으므로 훈련 전에 모든 훈련 데이터를 전처리할 수 있다.
- 하지만 전처리 과정을 바꿀 때마다 다른 코드를 수정해야하기 때문에 시간이 많이 걸리고, 에러를 만들기 쉽다. 이는 훈려/서빙 왜곡 (training/seving skew)은 버그나 성능 감소로 이어질 수 있다.
- 좀더 나은 방법은 훈련된 모델을 받아 앱이나 브라우저에 배포하기 전에 전처리를 담당하는 층을 동적으로 추가하는 것이다.
- 전처리 연산을 딱 한 번만 정의하기 위해 TF 변환(transform)이 만들어졌다.

```
try:
    import tensorflow_transform as tft

    def preprocess(inputs):  # inputs is a batch of input features
        median_age = inputs["housing_median_age"]
        ocean_proximity = inputs["ocean_proximity"]
        standardized_age = tft.scale_to_z_score(median_age - tft.mean(median_age))
        ocean_proximity_id = tft.compute_and_apply_vocabulary(ocean_proximity)
        return {
            "standardized_median_age": standardized_age,
            "ocean_proximity_id": ocean_proximity_id
        }
except ImportError:
    print("TF Transform is not installed. Try running: pip3 install -U tensorflow-transform")
```

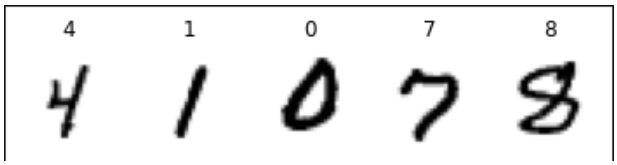
▼ 13.5 텐서플로 데이터셋(TFDS) 프로젝트

- 텐서플로 데이터셋을 사용하면 널리 사용하는 데이터셋을 쉽게 다운로드 할 수 있다.

```
import tensorflow_datasets as tfds

datasets = tfds.load(name="mnist")
mnist_train, mnist_test = datasets["train"], datasets["test"]

print(tfds.list_builders())
```

```
plt.figure(figsize=(6,3))
mnist_train = mnist_train.repeat(5).batch(32).prefetch(1)
for item in mnist_train:
    images = item["image"]
    labels = item["label"]
    for index in range(5):
        plt.subplot(1, 5, index + 1)
        image = images[index, ..., 0]
        label = labels[index].numpy()
        plt.imshow(image, cmap="binary")
        plt.title(label)
```

```
        plt.axis("off")
    break # just showing part of the first batch
```




```
datasets = tfds.load(name="mnist")
mnist_train, mnist_test = datasets["train"], datasets["test"]
mnist_train = mnist_train.repeat(5).batch(32)
mnist_train = mnist_train.map(lambda items: (items["image"], items["label"]))
mnist_train = mnist_train.prefetch(1)
for images, labels in mnist_train.take(1):
    print(images.shape)
    print(labels.numpy())
```

```
datasets = tfds.load(name="mnist", batch_size=32, as_supervised=True)
mnist_train = datasets["train"].repeat().prefetch(1)
model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28, 1]),
    keras.layers.Lambda(lambda images: tf.cast(images, tf.float32)),
    keras.layers.Dense(10, activation="softmax")])
model.compile(loss="sparse_categorical_crossentropy",
              optimizer=keras.optimizers.SGD(lr=1e-3),
              metrics=["accuracy"])
model.fit(mnist_train, steps_per_epoch=60000 // 32, epochs=5)
```

- 텐서플로 허브

```
#텐서플로 허브
import tensorflow_hub as hub

hub_layer = hub.KerasLayer("https://tfhub.dev/google/tf2-preview/nnlm-en-dim50/1",
                           output_shape=[50], input_shape=[], dtype=tf.string)

model = keras.Sequential()
model.add(hub_layer)
model.add(keras.layers.Dense(16, activation='relu'))
model.add(keras.layers.Dense(1, activation='sigmoid'))

model.summary()
```

```
sentences = tf.constant(["It was a great movie", "The actors were amazing"])
embeddings = hub_layer(sentences)

embeddings
```