Fəsil 13. TensorFlow ilə Məlumatın Yüklənməsi və Əvvəlcədən Emalı

2-ci fəsildə gördünüz ki, məlumatın yüklənməsi və əvvəlcədən emalı hər bir maşın öyrənməsi layihəsinin vacib hissəsidir. Siz Pandas kitabxanasından istifadə edərək (dəyişdirilmiş) Kaliforniya daşınmaz əmlak məlumat dəstini (CSV faylında saxlanmış) yükləyib araşdırdınız və əvvəlcədən emal üçün Scikit-Learn-un transformerlərindən istifadə etdiniz. Bu alətlər olduqca əlverişlidir və xüsusilə məlumatlarla işləyib eksperimentlər apararkən onlardan tez-tez istifadə edəcəksiniz.

Lakin, TensorFlow modellərini böyük məlumat dəstləri üzərində öyrədərkən TensorFlow-un öz məlumat yükləmə və əvvəlcədən emal API-si olan **tf.data**-dan istifadə etməyə üstünlük verə bilərsiniz. Bu API məlumatı çox effektiv şəkildə yükləyib əvvəlcədən emal edə bilir, multithreading və növbələmə vasitəsilə eyni anda bir neçə fayldan oxuma, nümunələrin qarışdırılması və batch-lərə bölünməsi və daha çoxunu edə bilir. Üstəlik, bunu dinamik şəkildə həyata keçirir—GPU və ya TPU-lar hazırkı batch-i öyrədərkən, məlumatın növbəti batch-i bir neçə CPU nüvəsi üzərində yüklənir və əvvəlcədən emal edilir.

tf.data API yaddaşa sığmayan məlumat dəstlərini idarə etməyə imkan verir və hardware resurslarından tam istifadə edərək təlim prosesini sürətləndirir. Bu API standart olaraq mətn fayllarından (CSV fayllar kimi), sabit ölçülü qeydlər olan ikili fayllardan və dəyişən ölçülü qeydləri dəstəkləyən TensorFlow-un TFRecord formatından oxuya bilir. TFRecord, adətən açıq mənbəli ikili format olan **protocol buffers** daxil edən çevik və effektiv bir formatdır. **tf.data** API həmçinin SQL verilənlər bazalarından oxumağı dəstəkləyir. Bundan əlavə, Google-un BigQuery xidməti kimi müxtəlif məlumat mənbələrindən oxumağa imkan verən bir çox açıq mənbə genişləndirmələri də mövcuddur (bax: https://tensorflow.org/io).

Keras, həmçinin güclü və istifadəsi asan preprocessing qatları ilə gəlir. Bu qatlar modellərinizə daxil edilə bilər, beləliklə, modeli istehsalata yerləşdirərkən xam məlumatları birbaşa qəbul edə bilər. Bunun üçün əlavə əvvəlcədən emal kodu yazmağınıza ehtiyac qalmır. Bu, təlim zamanı istifadə olunan əvvəlcədən emal kodu ilə istehsalatda istifadə olunan əvvəlcədən emal kodu arasında uyğunsuzluq riskini aradan qaldırır ki, bu da çox vaxt təlim/istismar uyğunsuzluğuna səbəb ola bilər. Əgər modelinizi müxtəlif proqramlaşdırma dillərində yazılmış tətbiqlərdə yerləşdirirsinizsə, eyni əvvəlcədən emal kodunu bir neçə dəfə yenidən yazmaq məcburiyyətində qalmayacaqsınız, bu da uyğunsuzluq riskini azaldır.

Gördüyünüz kimi, hər iki API birgə istifadə edilə bilər—məsələn, **tf.data**-nın təklif etdiyi effektiv məlumat yükləmənin və Keras-ın preprocessing qatlarının rahatlığından faydalanmaq üçün.

Bu fəsildə əvvəlcə **tf.data** API-ni və TFRecord formatını əhatə edəcəyik. Daha sonra Keras preprocessing qatlarını və onların **tf.data** API ilə necə istifadə olunacağını araşdıracağıq. Nəhayət, TensorFlow Datasets və TensorFlow Hub kimi məlumat yükləmə və əvvəlcədən emal üçün faydalı ola biləcək bir neçə əlaqəli kitabxanaya qısa bir nəzər salacağıq. Elə isə başlayaq!

`tf.data` API

`tf.data` API-nin mərkəzində **`tf.data.Dataset`** anlayışı dayanır: bu, bir məlumat maddələri ardıcıllığını təmsil edir. Adətən, diskdən tədricən məlumat oxuyan datasetlərdən istifadə edirsiniz, lakin sadəlik üçün bir tensoru **`tf.data.Dataset.from_tensor_slices()`** funksiyası ilə datasetə çevirək:

```
import tensorflow as tf
X = tf.range(10)  # istənilən məlumat tensörü
dataset = tf.data.Dataset.from_tensor_slices(X)
dataset
# Çıxış: <TensorSliceDataset shapes: (), types: tf.int32>
```

`from_tensor_slices()` funksiyası bir tensor götürür və onun birinci ölçüsü üzrə hər bir hissəni əhatə edən bir dataset yaradır. Yuxarıdakı misalda dataset 10 elementdən ibarətdir: tensorlar 0, 1, 2, ..., 9. Eyni datasetə **`tf.data.Dataset.range(10)`** funksiyası ilə də nail ola bilərdik (fərq ondadır ki, elementlər 32-bit deyil, 64-bit tam ədədlər olardı).

Datasetin elementlərini sadəcə iterasiya edərək yoxlaya bilərsiniz:

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

```
Output :
tf.Tensor(0, shape=(), dtype=int32)
tf.Tensor(1, shape=(), dtype=int32)
[...]
tf.Tensor(9, shape=(), dtype=int32)
```

Qeyd:

`tf.data` API bir **axın API**-dir: datasetin elementlərini çox effektiv iterasiya edə bilərsiniz, lakin indeksləmə və ya hissələrə ayırma üçün nəzərdə tutulmayıb.

Datasetlər, tensor cütlüklərindən, ad/tensor cütlərindən ibarət lüğətlərdən və ya iç-içə quruluşlardan da ibarət ola bilər. Struktur daxilindəki tensorları dilimləyərkən həmin struktur (tuple və ya dictionary) qorunur. Məsələn:

```
X_nested = {"a": ([1, 2, 3], [4, 5, 6]), "b": [7, 8, 9]}
dataset = tf.data.Dataset.from_tensor_slices(X_nested)
for item in dataset:
        print(item)
```

```
Output :
{'a': (<tf.Tensor: [...] = 1>, <tf.Tensor: [...] = 4>), 'b': <tf.Tensor: [...] = 7>}
{'a': (<tf.Tensor: [...] = 2>, <tf.Tensor: [...] = 5>), 'b': <tf.Tensor: [...] = 8>}
{'a': (<tf.Tensor: [...] = 3>, <tf.Tensor: [...] = 6>), 'b': <tf.Tensor: [...] = 9>}
```

Transformasiyaların Zəncirlənməsi

Dataset yaradıldıqdan sonra, onun üzərində müxtəlif transformasiyalar tətbiq edə bilərsiniz. Datasetin hər transformasiya metodu yeni bir dataset qaytarır, beləliklə, transformasiyaları zəncirləyə bilərsiniz. Aşağıdakı nümunəyə baxaq:

```
dataset = tf.data.Dataset.from_tensor_slices(tf.range(10))
dataset = dataset.repeat(3).batch(7)
for item in dataset:
        print(item)
```

```
Output :
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)
```

Bu nümunədə:

1. **`repeat(3)`** metodu çağırılır və ilkin datasetin elementlərini üç dəfə təkrarlayan yeni bir dataset yaradılır. Bu, məlumatı yaddaşda üç dəfə təkrarlamır! Əgər bu metodu arqumentsiz çağırsanız, dataset sonsuz dəfə təkrarlanacaq.
2. Daha sonra **`batch(7)`** metodu çağırılır, bu da əvvəlki datasetin elementlərini yeddi maddədən ibarət qruplara bölən yeni bir dataset yaradır.

Beləliklə, transformasiyaları zəncirləyərək məlumat üzərində daha mürəkkəb emal prosesləri yarada bilərsiniz.

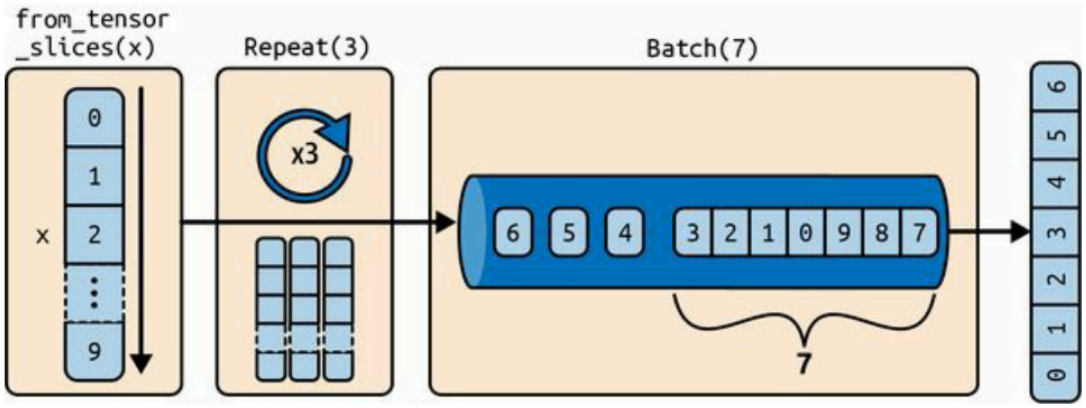

Figure 13-1. Chaining dataset transformations

Nəhayət, bu son datasetin elementləri üzərində iterasiya edirik. **`batch()`** metodu son batch-i yeddi əvəzinə iki elementlə çıxarmalı idi, lakin bütün batch-lərin eyni ölçüdə olmasını istəyirsinizsə, **`batch(drop_remainder=True)`** metodunu çağıraraq bu son batch-i ata bilərsiniz.

XƏBƏRDARLIQ
Dataset metodları datasetləri dəyişdirmir — onlar yenilərini yaradır. Buna görə də bu yeni datasetlərə bir istinad saxladığınızdan əmin olun (məsələn, **`dataset = ...`** ilə), əks halda heç nə baş verməyəcək.

Dataset elementlərini transformasiya etmək üçün **`map()`** metodunu da çağırmaq mümkündür. Məsələn, bu, bütün batch-ləri iki ilə vuraraq yeni bir dataset yaradır:

```
dataset = dataset.map(lambda x: x * 2)  # x bir batch-dir
for item in dataset:
        print(item)
```

```
Output :
tf.Tensor([ 0  2  4  6  8 10 12], shape=(7,), dtype=int32)
tf.Tensor([14 16 18  0  2  4  6], shape=(7,), dtype=int32)
[...]
```

`map()` metodu məlumatlar üzərində əvvəlcədən işləmə (preprocessing) tətbiq etmək üçün istifadə olunur. Bəzən bu proseslər kifayət qədər intensiv hesablamaları, məsələn, şəkli yenidən ölçüləndirmə və ya fırlatma kimi əməliyyatları əhatə edə bilər. Bu hallarda prosesi sürətləndirmək üçün çoxlu axınlar (threads) başlatmaq istəyə bilərsiniz. Bunun üçün **`num_parallel_calls`** arqumentini işləyən axınların sayına və ya **`tf.data.AUTOTUNE`**-ə təyin edə bilərsiniz. **`map()`** metoduna ötürdüyünüz funksiyanın TF funksiyasına çevrilə bilən olması vacibdir (bax: 12-ci fəsil).

Datasetləri sadəcə filtrasiya etmək də mümkündür və bu, **`filter()`** metodu vasitəsilə edilir. Məsələn, bu kod, yalnız cəmi 50-dən böyük olan batch-ləri saxlayan bir dataset yaradır:

```
dataset = dataset.filter(lambda x: tf.reduce_sum(x) > 50)
for item in dataset:
        print(item)
```

```
Output :
tf.Tensor([14 16 18  0  2  4  6], shape=(7,), dtype=int32)
tf.Tensor([ 8 10 12 14 16 18  0], shape=(7,), dtype=int32)
tf.Tensor([ 2  4  6  8 10 12 14], shape=(7,), dtype=int32)
```

Çox vaxt datasetin yalnız bir neçə elementinə baxmaq istəyirsiniz. Bunun üçün **`take()`** metodundan istifadə edə bilərsiniz:

```
for item in dataset.take(2):
        print(item)
```

```
Output :
tf.Tensor([14 16 18  0  2  4  6], shape=(7,), dtype=int32)
tf.Tensor([ 8 10 12 14 16 18  0], shape=(7,), dtype=int32)
```

Məlumatların Qarışdırılması

4-cü fəsildə müzakirə etdiyimiz kimi, gradient enişi (gradient descent) təlim dəstindəki nümunələr müstəqil və eyni paylanmış olduqda (IID) ən yaxşı şəkildə işləyir. Bunu təmin etməyin sadə yolu **`shuffle()`** metodundan istifadə edərək nümunələri qarışdırmaqdır. Bu metod mənbə datasetin ilk elementlərini bir buferə dolduraraq yeni bir dataset yaradır. Daha sonra ondan element tələb olunduqda, həmin buferdən təsadüfi bir element çəkir və onu mənbə datasetdən yeni bir elementlə əvəz edir. Bu proses mənbə dataset tamamilə iterasiya edilənə qədər davam edir. Bu nöqtədən sonra bufer boşalana qədər təsadüfi elementlər çıxarılır.

Bufer ölçüsünü təyin etmək vacibdir və onu kifayət qədər böyük etməlisiniz, əks halda qarışdırma effektiv olmayacaq. Amma yadda saxlayın ki, RAM miqdarınızı aşmayın; hətta kifayət qədər RAM-a malik olsanız belə, dataset ölçüsünü keçməyə ehtiyac yoxdur. Proqramınızı hər dəfə işlətdikdə eyni təsadüfi ardıcıllığı əldə etmək istəyirsinizsə, təsadüfi toxum (random seed) təyin edə bilərsiniz.

Məsələn, aşağıdakı kod 0-dan 9-a qədər olan ədədləri iki dəfə təkrarlayaraq bir dataset yaradır, bu dataset 4 ölçülü bir bufer və 42 təsadüfi toxumla qarışdırılır və 7 ölçülü batch-lər şəklində qruplaşdırılır:

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

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

MƏSLƏHƏT

Əgər **`repeat()`** metodunu qarışdırılmış bir dataset üzərində çağırsanız, varsayılan olaraq hər iterasiya zamanı yeni bir ardıcıllıq yaradılacaq. Bu, ümumiyyətlə yaxşı fikirdir, amma hər iterasiya zamanı eyni ardıcıllığı istifadə etmək istəyirsinizsə (məsələn, testlər və ya hata ayıklama üçün), **`shuffle()`** metodunu çağırarkən **`reshuffle_each_iteration=False`** olaraq təyin edə bilərsiniz.

RAM-a yerləşməyən böyük datasetlər üçün bu sadə qarışdırma-bufer yanaşması kifayət etməyə bilər, çünki bufer datasetə nisbətən kiçik olacaq. Bu problemi həll etmək üçün mənbə

məlumatını özünü qarışdırmaq olar (məsələn, Linux-da **`shuf`** əmri ilə mətn fayllarını qarışdırmaq mümkündür). Bu, qarışdırmanı xeyli yaxşılaşdıracaq!

Hətta mənbə məlumat qarışdırılmış olsa da, onu bir az daha qarışdırmaq istəyəcəksiniz, çünki hər epoxda eyni ardıcıllıq təkrarlanacaq və model təsadüfi bir nümunənin qaydasındakı bəzi yanlış təsadüfi naxışlar səbəbindən qərəzli ola bilər. Qarışdırmanı artırmaq üçün ümumi yanaşma mənbə məlumatını bir neçə fayla bölmək, sonra təlim zamanı onları təsadüfi qaydada oxumaqdır. Ancaq eyni faylda yerləşən nümunələr hələ də bir-birinə yaxın olacaq. Bunun qarşısını almaq üçün bir neçə faylı təsadüfi şəkildə seçə və onların qeydlərini paralel olaraq oxuyaraq birləşdirə bilərsiniz. Bunun üzərinə **`shuffle()`** metodundan istifadə edərək əlavə bir qarışdırma buferi əlavə edə bilərsiniz.

Bu çox iş kimi səslənsə də, narahat olmayın: **`tf.data`** API-si bütün bunları bir neçə kod sətiri ilə həyata keçirməyə imkan verir. Gəlin bunu necə edəcəyinizi nəzərdən keçirək.

Çoxsaylı Fayllardan Sətirlərin Qarışdırılması
Təsəvvür edək ki, Kaliforniya mənzil datasetini yükləmisiniz, onu qarışdırmısınız (əgər artıq qarışdırılmış deyildisə) və onu təlim, yoxlama və test dəstlərinə bölmüsünüz. Daha sonra hər bir dəsti çoxsaylı CSV fayllarına bölmüsünüz. Hər bir fayl belə görünür (hər sətir səkkiz giriş xüsusiyyətini və hədəf median mənzil dəyərini ehtiva edir):

```
MedInc,HouseAge,AveRooms,AveBedrms,Popul…,AveOccup,Lat…,Long…,MedianHouseV
alue
3.5214,15.0,3.050,1.107,1447.0,1.606,37.63,-122.43,1.442
5.3275,5.0,6.490,0.991,3464.0,3.443,33.69,-117.39,1.687
3.1,29.0,7.542,1.592,1328.0,2.251,38.44,-122.98,1.621
[...]
```

Tutaq ki, **`train_filepaths`** dəyişənində təlim fayllarının yollarını saxlamaq üçün bir siyahınız var (və sizdə **`valid_filepaths`** və **`test_filepaths`** də var):

```
train_filepaths = [
        'datasets/housing/my_train_00.csv',
        'datasets/housing/my_train_01.csv',
        ...
]
```

Alternativ olaraq fayl şablonlarından istifadə edə bilərsiniz, məsələn:

```
train_filepaths = "datasets/housing/my_train_*.csv"
```

İndi yalnız bu fayl yollarını ehtiva edən bir dataset yaradaq:

```
filepath_dataset = tf.data.Dataset.list_files(train_filepaths, seed=42)
```

Varsayılan olaraq, **`list_files()`** funksiyası fayl yollarını qarışdırır. Bu, ümumiyyətlə faydalıdır, lakin hər hansı bir səbəbdən qarışdırmaq istəmirsinizsə, **`shuffle=False`** parametri təyin edə bilərsiniz.

Sonra **`interleave()`** metodunu çağıraraq beş fayldan bir dəfə oxuyub onların sətirlərini qarışdıra bilərsiniz. Hər faylın başlıq sətrini (header) keçmək üçün **`skip()`** metodundan istifadə edə bilərsiniz:

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

`interleave()` metodu, **`filepath_dataset`**-dən beş fayl yolu çəkəcək və hər biri üçün göstərdiyiniz funksiyanı (bu nümunədə bir lambda funksiyası) çağıraraq yeni bir dataset yaradacaq (bu halda **`TextLineDataset`**).

Bu mərhələdə cəmi yeddi dataset olacaq:

1. Fayl yollarını saxlayan **filepath_dataset**,
2. Qarışdırılmış **interleave_dataset**,
3. **`interleave_dataset`** tərəfindən daxildə yaradılan beş **TextLineDataset**.

`interleave_dataset` üzərində iterasiya etdikdə, bu beş **TextLineDataset**-in hər birindən bir sətir oxuyacaq, bütün sətirlər tükənənə qədər davam edəcək. Daha sonra **filepath_dataset**-dən növbəti beş fayl yolunu çəkəcək və eyni şəkildə qarışdırmağa davam edəcək.

Mümkün qədər yaxşı nəticə əldə etmək üçün faylların uzunluqlarının eyni olması tövsiyə olunur. Əks halda, ən uzun faylın sonundakı sətirlər qarışdırılmayacaq.

Parallel Oxuma

Varsayılan olaraq **`interleave()`**, paralel işləmədən fayllardan sətirləri bir-bir oxuyur. Əgər faylları paralel oxumaq istəyirsinizsə, **`interleave()`** metodunun **`num_parallel_calls`** parametrini təyin edə bilərsiniz. Məsələn, **`tf.data.AUTOTUNE`** təyin edərək TensorFlow-un CPU-nun vəziyyətinə əsasən uyğun sayda paralel iş ipini avtomatik seçməsini təmin edə bilərsiniz.

Datasetdə nələrin olduğunu görə bilərsiniz:

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

```
Output :
tf.Tensor(b'4.5909,16.0,[...],33.63,-117.71,2.418', shape=(), dtype=string)
tf.Tensor(b'2.4792,24.0,[...],34.18,-118.38,2.0', shape=(), dtype=string)
tf.Tensor(b'4.2708,45.0,[...],37.48,-122.19,2.67', shape=(), dtype=string)
tf.Tensor(b'2.1856,41.0,[...],32.76,-117.12,1.205', shape=(), dtype=string)
```

```
tf.Tensor(b'4.1812,52.0,[...],33.73,-118.31,3.215', shape=(), dtype=string)
```

Bunlar beş təsadüfi CSV faylından gələn ilk sətirlərdir (başlıq sətri nəzərə alınmadan). Gözəl görünür!

Qeyd:
`TextLineDataset` konstruktoruna fayl yollarının siyahısını ötürmək mümkündür: bu zaman dataset hər faylı ardıcıllıqla, sətir-sətir oxuyacaq. Əgər **`num_parallel_reads`** arqumentini birdən böyük bir dəyərə təyin etsəniz, dataset həmin sayda faylı paralel oxuyacaq və onların sətirlərini qarışdıracaq (qarışdırma üçün **`interleave()`** metodunu çağırmağa ehtiyac yoxdur). Bununla belə, fayllar qarışdırılmayacaq və başlıq sətirləri keçilməyəcək.

Məlumatların Ön İşlənməsi

Artıq hər bir nümunəni bayt sətiri şəklində qaytaran bir mənzil datasetimiz var. Biz sətirləri təhlil etmək və məlumatları miqyaslandırmaq daxil olmaqla, bəzi ön işləmə işləri görməliyik. Gəlin bu ön işləməni həyata keçirəcək xüsusi funksiyaları implementasiya edək:

```
X_mean, X_std = [...]  # Təlim setində hər xüsusiyyətin orta və standart sapması
n_inputs = 8

def parse_csv_line(line):
        defs = [0.] * n_inputs + [tf.constant([], dtype=tf.float32)]
        fields = tf.io.decode_csv(line, record_defaults=defs)
        return tf.stack(fields[:-1]), tf.stack(fields[-1:])

def preprocess(line):
        x, y = parse_csv_line(line)
        return (x - X_mean) / X_std, y
```

Gəlin bu kodun addımlarını izah edək:

1. Kod təlim setindəki hər xüsusiyyətin **orta** və **standart sapmasını** əvvəlcədən hesablamağımızı nəzərdə tutur. **X_mean** və **X_std**, hər biri səkkiz ədəddən ibarət olan 1D tensorlardır (və ya NumPy massivləri). Bunları datasetin kifayət qədər böyük təsadüfi nümunəsi üzərində Scikit-Learn **StandardScaler** ilə hesablamaq mümkündür. Bu bölmənin sonrakı hissəsində Keras-ın ön işləmə qatından istifadə edəcəyik.
2. **parse_csv_line()** funksiyası bir CSV sətrini qəbul edir və onu təhlil edir. Bunun üçün, **tf.io.decode_csv()** funksiyasından istifadə olunur. Bu funksiya iki arqument qəbul edir:
    - Təhlil ediləcək sətir.
    - Hər sütunun CSV-dəki standart dəyərlərini ehtiva edən massiv (**defs**). Bu massiv TensorFlow-a yalnız hər sütunun standart dəyərini deyil, həm də sütunların sayını və onların tiplərini bildirir. Bu nümunədə bütün xüsusiyyət sütunlarının tipinin float olduğunu və itkin dəyərlərin 0-a dəyişdirilməli

olduğunu bildiririk. Hədəf sütun (son sütun) üçün isə boş bir **tf.float32** massivindən istifadə edirik. Bu TensorFlow-a bu sütunun float olduğunu, lakin standart dəyərin olmadığını bildirir. Beləliklə, əgər bu sütunda itkin dəyər olarsa, istisna atılacaq.

3. **tf.io.decode_csv()** funksiyası sütun başına bir scalar tensorun siyahısını qaytarır. Ancaq biz 1D tensor massivi qaytarmalıyıq. Bunun üçün son sütun (hədəf) istisna olmaqla digər tensorları **tf.stack()** funksiyası ilə 1D massivə yığırıq. Sonra isə hədəf dəyəri üçün də eyni şeyi edirik: bu, tək bir dəyərdən ibarət olan 1D tensor yaradır. Beləliklə, **parse_csv_line()** funksiyası xüsusiyyətləri və hədəf dəyərini qaytarır.
4. Nəhayət, xüsusi **preprocess()** funksiyası **parse_csv_line()** funksiyasını çağırır, xüsusiyyətləri miqyaslandırır (orta dəyərləri çıxır və standart sapmalara bölür) və miqyaslandırılmış xüsusiyyətlərdən və hədəfdən ibarət tuple qaytarır.

Bu ön işləmə funksiyasını test edək:

```
>>> preprocess(b'4.2083,44.0,5.3232,0.9171,846.0,2.3370,37.47,-122.2,2.782')
(<tf.Tensor: shape=(8,), dtype=float32, numpy=
array([ 0.16579159,  1.216324 , -0.05204564, -0.39215982, -0.5277444 ,
        -0.2633488 ,  0.8543046 , -1.3072058 ], dtype=float32)>,
<tf.Tensor: shape=(1,), dtype=float32, numpy=array([2.782], dtype=float32)>)
```

Görünür hər şey qaydasındadır! **preprocess()** funksiyası bir nümunəni bayt sətrindən gözəl miqyaslandırılmış tensor şəklinə və ona uyğun etiketi çevirə bilir. İndi datasetin **map()** metodundan istifadə edərək bu funksiyanı hər bir nümunəyə tətbiq edə bilərik.

Hər Şeyi Bir Araya Toplamaq

Kodu daha yenidən istifadə edilə bilən etmək üçün, indiyə qədər müzakirə etdiyimiz hər şeyi birləşdirib bir yardımçı funksiyada toplayaq. Bu funksiya, bir neçə CSV faylından **California housing** məlumatlarını effektiv şəkildə yükləyəcək, ön işləmə edəcək, təsadüfi şəkildə qarışdıracaq və batch-lərə böləcək bir dataset yaradacaq və qaytaracaq (Şəkil 13-2-ə baxın):

```
def csv_reader_dataset(filepaths, n_readers=5, n_read_threads=None,
                       n_parse_threads=5, shuffle_buffer_size=10_000, seed=42,
                       batch_size=32):
    dataset = tf.data.Dataset.list_files(filepaths, seed=seed)
    dataset = dataset.interleave(
    lambda filepath: tf.data.TextLineDataset(filepath).skip(1),
    cycle_length=n_readers, num_parallel_calls=n_read_threads)
    dataset = dataset.map(preprocess, num_parallel_calls=n_parse_threads)
    dataset = dataset.shuffle(shuffle_buffer_size, seed=seed)
    return dataset.batch(batch_size).prefetch(1)
```

Son sətirdə **prefetch()** metodundan istifadə etdiyimizə diqqət yetirin. Bu, performans baxımından vacibdir və indi bunun əhəmiyyətini görəcəksiniz.

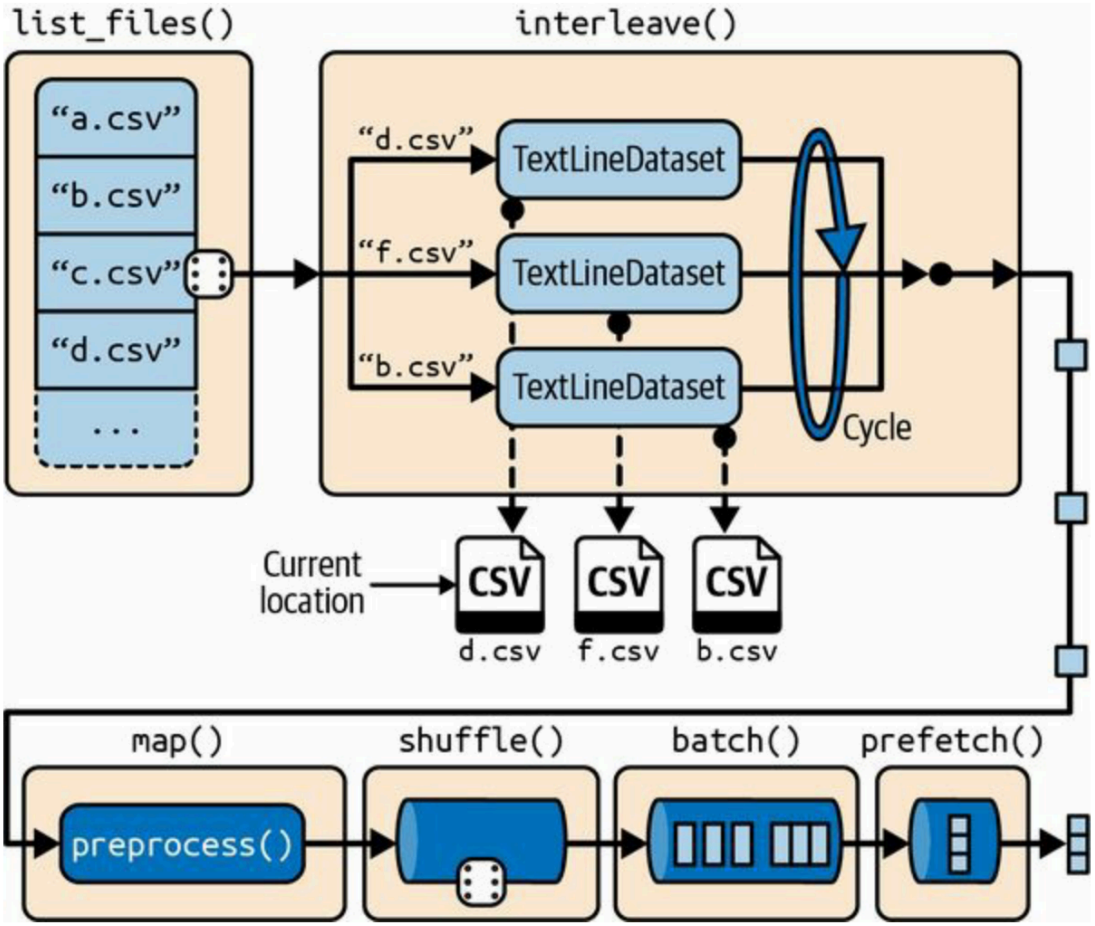

Figure 13-2. Loading and preprocessing data from multiple CSV files

Prefetching

csv_reader_dataset() funksiyasının sonunda **prefetch(1)** çağırışı edərək, həmişə bir batch-i qabaqcadan hazırlamağa çalışan bir dataset yaradırıq. Başqa sözlə, təlim alqoritmimiz bir batch üzərində işləyərkən, dataset paralel olaraq növbəti batch-i hazırlamaq (məsələn, disklə məlumat oxumaq və ön işləmə aparmaq) üzərində işləyəcək. Bu, performansı əhəmiyyətli dərəcədə yaxşılaşdıra bilər, bunu Şəkil 13-3-də görmək mümkündür.

Əgər məlumatın yüklənməsi və ön işləmənin çox axınlı (multithreaded) olmasını təmin etsək (bunun üçün **interleave()** və **map()** çağırarkən **num_parallel_calls** parametrini təyin edirik), bir batch məlumatın hazırlanmasını GPU-da bir təlim addımının yerinə yetirilməsindən daha qısa vaxtda tamamlayaraq çoxsaylı CPU nüvələrindən istifadə edə bilərik. Bu şəkildə GPU demək olar ki, tam olaraq (%100) istifadə olunacaq (CPU-dan GPU-ya məlumat ötürmə vaxtı istisna olmaqla), və təlim daha sürətli həyata keçiriləcək.

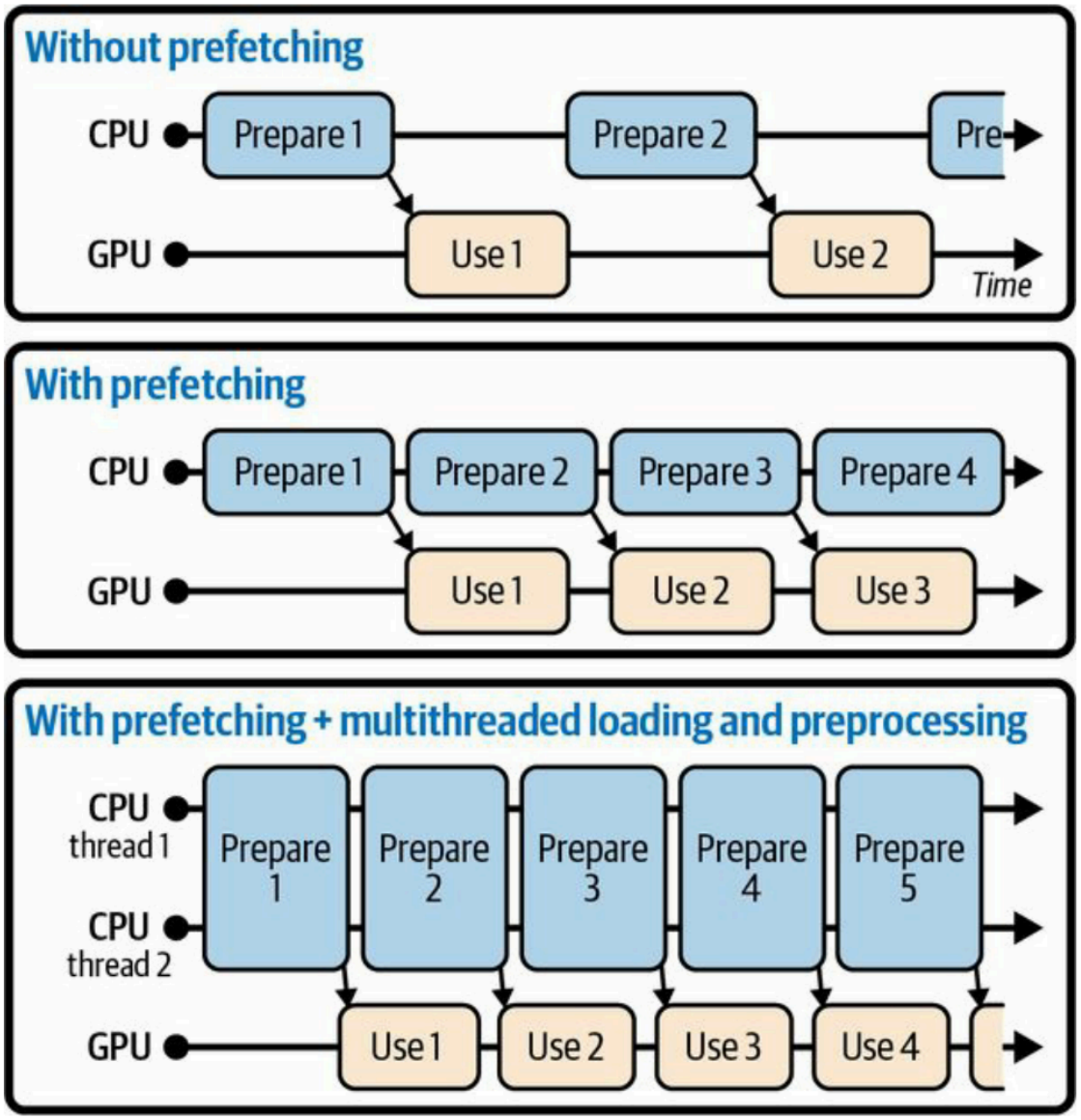

Figure 13-3. With prefetching, the CPU and the GPU work in parallel: as the GPU works on one batch, the CPU works on the next

TIP

Əgər GPU kartı almayı planlaşdırırsınızsa, onun emal gücü və yaddaş həcmi əlbəttə ki, çox vacibdir (xüsusilə böyük kompüter görmə və ya təbii dil işləmə modelləri üçün böyük miqdarda RAM olduqca əhəmiyyətlidir). Yaxşı performans üçün GPU-nun yaddaş bant genişliyi də eyni dərəcədə vacibdir; bu, onun RAM-a saniyədə daxil ola və ya RAM-dan çıxa biləcəyi gigabayt məlumatın miqdarıdır.

Əgər dataset kifayət qədər kiçikdirsə və yaddaşa sığırsa, **cache()** metodunu istifadə edərək dataset-in məzmununu RAM-da saxlamaqla təlimi əhəmiyyətli dərəcədə sürətləndirə bilərsiniz. Ümumiyyətlə, bunu məlumatları yüklədikdən və ön işlədikdən sonra, lakin qarışdırmadan, təkrarlamadan, batch-lamadan və prefetching-dən əvvəl etməlisiniz. Bu

şəkildə, hər bir nümunə yalnız bir dəfə oxunacaq və ön işlənəcək (hər epoxda bir dəfə deyil), lakin məlumat hər epoxda fərqli qarışdırılacaq və növbəti batch hələ də əvvəlcədən hazırlanacaq.

İndi siz çoxsaylı mətn fayllarından məlumatları yükləmək və ön işləmək üçün effektiv giriş boru xətləri qurmağı öyrəndiniz. Ən çox istifadə olunan dataset metodlarını müzakirə etdik, amma nəzərdən keçirmək istəyə biləcəyiniz bir neçə başqa metod da mövcuddur, məsələn, **concatenate()**, **zip()**, **window()**, **reduce()**, **shard()**, **flat_map()**, **apply()**, **unbatch()** və **padded_batch()**. Həmçinin, **from_generator()** və **from_tensors()** kimi bir neçə əlavə sinif metodları mövcuddur, bunlar sırasıyla Python generator-dən və ya tensorlar siyahısından yeni bir dataset yaradır. Ətraflı məlumat üçün API sənədlərini yoxlayın. Həmçinin qeyd edin ki, **tf.data.experimental** daxilində eksperimental xüsusiyyətlər mövcuddur, bunlardan çoxu gələcək buraxılışlarda əsas API-yə daxil olacaq (məsələn, **CsvDataset** sinfini və hər bir sütunun növünü müəyyənləşdirən **make_csv_dataset()** metodunu yoxlayın).

Using the Dataset with Keras

İndi əvvəlki yazdığımız **csv_reader_dataset()** funksiyasını istifadə edərək, təlim seti, doğrulama seti və test seti üçün dataset yarada bilərik. Təlim seti hər epoxda qarışdırılacaq (diqqət yetirin ki, doğrulama və test setləri də qarışdırılacaq, baxmayaraq ki, buna ehtiyacımız yoxdur):

```
train_set = csv_reader_dataset(train_filepaths)

valid_set = csv_reader_dataset(valid_filepaths)

test_set = csv_reader_dataset(test_filepaths)
```

İndi bu dataset-ləri istifadə edərək sadəcə bir Keras modeli qura və təlim edə bilərsiniz. Modelin **fit()** metodunu çağırarkən, **train_set**-i **X_train**, **y_train** əvəzinə verirsiniz və **validation_data=valid_set**-i **validation_data=(X_valid, y_valid)** əvəzinə verirsiniz. **fit()** metodu, hər epoxda təlim datasetini təkrarlamağı, hər epoxda fərqli təsadüfi ardıcıllıq istifadə edərək təmin edəcək:

```
model = tf.keras.Sequential([...])

model.compile(loss="mse", optimizer="sgd")

model.fit(train_set, validation_data=valid_set, epochs=5)
```

Eynilə, **evaluate()** və **predict()** metodlarına dataset verə bilərsiniz:

```
test_mse = model.evaluate(test_set)

new_set = test_set.take(3)  # 3 yeni nümunəmiz olduğunu təsəvvür edirik
```

```
y_pred = model.predict(new_set)  # və ya sadəcə NumPy array-i verə bilərsiniz
```

Digər setlərdən fərqli olaraq, **new_set** adətən etiketləri ehtiva etməyəcəkdir. Əgər varsa, burada olduğu kimi, Keras onları nəzərə almayacaq. Diqqət yetirin ki, bütün bu hallarda, hələ də dataset-lər əvəzinə NumPy array-lərdən istifadə edə bilərsiniz, əgər üstünlük verirsinizsə (lakin əlbəttə ki, əvvəlcə yüklənib ön işlənməlidirlər).

Əgər öz xüsusi təlim dövrünüzü qurmaq istəyirsinizsə (12-ci fəsildə müzakirə olunduğu kimi), sadəcə təlim seti üzərində təbii olaraq iterasiya edə bilərsiniz:

```
n_epochs = 5

for epoch in range(n_epochs):

        for X_batch, y_batch in train_set:

        [...]  # Bir gradient descent addımını yerinə yetirin
```

Əslində, modelin bütöv bir epox üçün təlim keçməsini təmin edən bir TF funksiyası yaratmaq mümkündür (12-ci fəsildə baxın). Bu, təlimi sürətləndirə bilər:

```
@tf.function

def train_one_epoch(model, optimizer, loss_fn, train_set):

        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))


optimizer = tf.keras.optimizers.SGD(learning_rate=0.01)

loss_fn = tf.keras.losses.mean_squared_error

for epoch in range(n_epochs):

        print("\rEpoch {}/{}".format(epoch + 1, n_epochs), end="")

        train_one_epoch(model, optimizer, loss_fn, train_set)
```

Keras-da **compile()** metodunun **steps_per_execution** arqumenti, **fit()** metodunun hər çağırışında təlim üçün istifadə etdiyi **tf.function**-da emal olunan batch sayını təyin etməyə imkan verir. Defolt olaraq bu dəyər 1-dir, ona görə də bunu 50-ə təyin etsəniz, performansda əhəmiyyətli bir yaxşılaşma görə bilərsiniz. Lakin, Keras callback-larının **on_batch_*()** metodları yalnız hər 50 batch-dən sonra çağırılacaq.

Təbrik edirəm, indi **tf.data API**-ni istifadə edərək güclü giriş boru xətləri qurmağı öyrəndiniz! Lakin, indiyə qədər **CSV** faylları istifadə etdik, bunlar ümumiyyətlə sadə və rahat olsa da, çox effektiv deyillər və böyük və ya kompleks məlumat strukturlarını (məsələn, şəkillər və ya audio) yaxşı dəstəkləmir. İndi **TFRecords** istifadə etməyə baxaq.

TIP

Əgər **CSV** fayllarından (və ya istifadə etdiyiniz digər formatlardan) məmnunsunuzsa, **TFRecords** istifadə etməyə ehtiyacınız yoxdur. Atalar sözünü deyək, əgər işləmirəmsə, düzəltməyin! **TFRecords** təlim zamanı yükləmə və parsing əməliyyatlarının dar boğaz olduğu hallarda faydalıdır.

TFRecord Formatı

TFRecord formatı, TensorFlow-un böyük miqdarda məlumatları saxlamaq və effektiv şəkildə oxumaq üçün üstünlük verdiyi formattır. Bu, çox sadə bir ikili formattır və yalnız müxtəlif ölçülərdəki ikili qeydlərdən ibarətdir (hər bir qeyd bir uzunluqdan, uzunluğun pozulmadığını yoxlamaq üçün CRC checksum-dan, sonra isə faktiki məlumatlardan və nəhayət məlumat üçün bir CRC checksum-dan ibarətdir).

TFRecord faylını **tf.io.TFRecordWriter** sinfini istifadə edərək asanlıqla yarada bilərsiniz:

```
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")
```

Sonra bir və ya bir neçə **TFRecord** faylını oxumaq üçün **tf.data.TFRecordDataset** istifadə edə bilərsiniz:

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

Bu, aşağıdakı nəticəni verəcək:

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

TIP

Defolt olaraq, **TFRecordDataset** faylları tək-tək oxuyacaq, lakin ona **filepaths** siyahısını ötürərək və **num_parallel_reads** parametrini bir rəqəm daha böyük olan bir dəyərə təyin

edərək faylları paralel oxumağa və qeydlərini birləşdirməyə imkan verə bilərsiniz. Alternativ olaraq, əvvəlki kimi bir neçə **CSV** faylını oxumaq üçün **list_files()** və **interleave()** istifadə edərək eyni nəticəni əldə edə bilərsiniz.

Sıxılmış TFRecord Faylları

Bəzən TFRecord fayllarınızı sıxmaq faydalı ola bilər, xüsusilə də şəbəkə əlaqəsi vasitəsilə yüklənmələri lazım olduqda. **TFRecord** faylını sıxılmış şəkildə yaratmaq üçün **options** arqumentini təyin edərək bunu edə bilərsiniz:

```
options = tf.io.TFRecordOptions(compression_type="GZIP")
with tf.io.TFRecordWriter("my_compressed.tfrecord", options) as f:
        f.write(b"Compress, compress, compress!")
```

Bir sıxılmış **TFRecord** faylını oxuyarkən, sıxma növünü göstərməlisiniz:

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

Protokol Buffers-a Qısa Giriş

Hər bir qeyd istədiyiniz hər hansı bir ikili formatı istifadə edə bilsə də, **TFRecord** faylları adətən seriyalaşdırılmış protokol buffers (həmçinin protobufs adlanır) ehtiva edir. Bu, Google tərəfindən 2001-ci ildə inkişaf etdirilən, 2008-ci ildə açıq mənbəyə çevrilən və indi geniş istifadə olunan, portativ, genişlənə bilən və effektiv bir ikili formattır. Xüsusilə gRPC-də, Google-ın uzaqdan prosedur çağırışı sistemində istifadə olunur. Onlar aşağıdakı kimi sadə bir dildə müəyyən edilir:

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

Bu protobuf tərifində, protobuf formatının 3-cü versiyasını istifadə etdiyimiz bildirilir və hər bir **Person** obyekti (isteğe bağlı olaraq) bir **name** (string tipində), bir **id** (int32 tipində) və sıfır və ya daha çox **email** sahəsinə (hər biri string tipində) sahib ola bilər. 1, 2 və 3 rəqəmləri sahə identifikatorlarıdır: bunlar hər bir qeydin ikili təmsilində istifadə olunacaq. Bir .proto faylında tərifiniz olduqda, onu tərtib edə bilərsiniz. Bu, Python-da (və ya digər dillərdə) giriş sinifləri yaratmaq üçün **protoc** adlı protobuf tərtibçisini tələb edir. Diqqət yetirin ki, TensorFlow-da ümumiyyətlə istifadə edəcəyiniz protobuf tərifləri artıq sizin üçün tərtib edilib və onların Python sinifləri TensorFlow kitabxanasının bir hissəsidir, beləliklə **protoc** istifadə etməyə ehtiyac yoxdur. Bütün bilməli olduğunuz şey Python-da protobuf giriş siniflərini necə istifadə edəcəyinizdir. Əsasları izah etmək üçün gəlin **Person** protobuf üçün yaradılmış giriş siniflərini istifadə edən sadə bir nümunəyə baxaq (şərhlərdə kod izah edilir):

```
>>> from person_pb2 import Person # yaradılmış giriş sinifini idxal et
>>> person = Person(name="Al", id=123, email=["a@b.com"]) # Person yaradın
```

```
>>> print(person) # Person-u göstərin
name: "Al"
id: 123
email: "a@b.com"
>>> person.name # bir sahəni oxuyun
'Al'
>>> person.name = "Alice" # bir sahəni dəyişdirin
>>> person.email[0] # təkrarlanan sahələr, massivlər kimi əldə edilə bilər
'a@b.com'
>>> person.email.append("c@d.com") # bir email ünvanı əlavə edin
>>> serialized = person.SerializeToString() # person-u bayt zəncirinə seriyalaşdırın
>>> serialized
b'\n\x05Alice\x10{\x1a\x07a@b.com\x1a\x07c@d.com'
>>> person2 = Person() # yeni bir Person yaradın
>>> person2.ParseFromString(serialized) # bayt zəncirini analiz edin (27 bayt uzunluğunda)
27
>>> person == person2 # indi onlar eynidir
True
```

Qısa olaraq, biz **protoc** tərəfindən yaradılmış **Person** sinifini idxal edirik, bir nümunə yaradırıq və onunla işləyirik, onu vizuallaşdırırıq, bəzi sahələri oxuyub yazırıq, sonra isə onu **SerializeToString()** metodu ilə seriyalaşdırırıq. Bu, şəbəkə üzərindən saxlamaq və ya göndərmək üçün hazır olan ikili məlumatdır. Bu ikili məlumatı oxuyarkən və ya aldıqda, biz **ParseFromString()** metodu ilə onu analiz edə bilərik və seriyalaşdırılmış obyektin bir surətini əldə edərik.

Seriyalaşdırılmış **Person** obyektini **TFRecord** faylında saxlaya bilər, sonra isə onu yükləyib analiz edə bilərsiniz: hər şey yaxşı işləyəcəkdir. Ancaq **ParseFromString()** TensorFlow əməliyyatı olmadığı üçün onu **tf.data** boru kəmərində (yalnız **tf.py_function()** əməliyyatında sarılaraq, bu da kodu yavaşlatacaq və daha az daşına bilən edəcək, baxmayaraq ki, 12-ci fəsildə bunu gördünüz) istifadə edə bilməzsiniz. Lakin siz **tf.io.decode_proto()** funksiyasını istifadə edə bilərsiniz, hansı ki, istədiyiniz hər hansı bir protobuf-u analiz edə bilər, təklif etdiyiniz protobuf tərifini versəniz (nümunə üçün notbuku yoxlaya bilərsiniz). Lakin praktikada TensorFlow-un xüsusi analiz əməliyyatları təqdim etdiyi əvvəlcədən təyin edilmiş protobuf-lardan istifadə etməyiniz daha yaxşı olacaq. İndi bu əvvəlcədən təyin edilmiş protobuf-lara baxaq.

TensorFlow Protobufs

TFRecord faylında ən çox istifadə olunan əsas protobuf **Example** protobuf-u olub, bu, verilənlər toplusunda bir nümunəni təmsil edir. O, adlandırılmış xüsusiyyətlərin siyahısını ehtiva edir, burada hər bir xüsusiyyət ya byte zəncirləri, ya da float nömrələri və ya tam ədədlərdən ibarət bir siyahı ola bilər. Aşağıda protobuf tərifi (TensorFlow-un mənbə kodundan) verilmişdir:

```
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; };
```

BytesList, **FloatList** və **Int64List** tərifləri kifayət qədər sadədir. Qeyd etmək lazımdır ki, **[packed = true]** təkrarlanan rəqəmli sahələr üçün daha səmərəli kodlaşdırma təmin etmək məqsədilə istifadə edilir. Bir **Feature** ya **BytesList**, ya **FloatList**, ya da **Int64List** ehtiva edir. **Features** (sonda "s" ilə) bir lüğət (dictionary) saxlayır ki, bu da xüsusiyyət adını müvafiq xüsusiyyət dəyərinə map edir. Nəhayət, **Example** yalnız **Features** obyektini ehtiva edir.

Qeyd:
Niyə **Example** tərif edilib, əgər o, yalnız bir **Features** obyektini ehtiva edir? Bəlkə TensorFlow inkişaf etdiriciləri bir gün ona daha çox sahə əlavə etməyi qərara alacaq. Yeni **Example** tərifi hələ də **features** sahəsini eyni ID ilə ehtiva etdikdə, geri uyğunluq təmin edilir. Bu genişlənə bilənlik protobuf-ların ən yaxşı xüsusiyyətlərindən biridir.

Aşağıda əvvəlki nümunədəki eyni şəxsi təmsil edən bir **tf.train.Example** necə yaradıla biləcəyinə baxaq:

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

person_example = Example(
        features=Features(
        feature={
        "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"]))
        }
        )
)
```

Kod bir az geniş və təkrarlayıcıdır, amma bunu kiçik bir köməkçi funksiyada asanlıqla yığa bilərsiniz. İndi biz **Example** protobuf-a sahibik, onu **SerializeToString()** metodu ilə seriyalaşdıra bilərik və sonra nəticə olaraq əldə olunan məlumatı **TFRecord** faylına yaza bilərik. Gəlin bunu beş dəfə yazaraq, bir neçə əlaqə məlumatımız olduğunu düşünək:

```
with tf.io.TFRecordWriter("my_contacts.tfrecord") as f:
    for _ in range(5):
        f.write(person_example.SerializeToString())
```

Adətən, beş **Example** yazmaqdan daha çox yazacaqsınız! Adətən, mövcud formatınızdan (məsələn, CSV faylları) oxuyan, hər bir nümunə üçün **Example** protobuf-u yaradan, onları seriyalaşdıran və bir neçə **TFRecord** faylına saxlayan bir konvertasiya skripti yaradacaqsınız. Bu prosesdə onları qarışdırmaq (shuffle) ən yaxşı təcrübədir. Bu müəyyən qədər iş tələb edir, buna görə bir daha əmin olun ki, bu, həqiqətən lazımdır (bəlkə boru kəməriniz CSV faylları ilə yaxşı işləyir).

İndi bir neçə seriyalaşdırılmış **Example** ehtiva edən gözəl bir **TFRecord** faylımız olduğuna görə, gəlin onu yükləməyə çalışaq.

Nümunələrin Yüklənməsi və Ayrılması

Seriyalaşdırılmış **Example** protobuf-larını yükləmək üçün bir daha **tf.data.TFRecordDataset** istifadə edəcəyik və hər bir **Example**-ı **tf.io.parse_single_example()** ilə ayrılacağıq. Bu, ən azı iki arqument tələb edir: seriyalaşdırılmış məlumatı ehtiva edən bir scalar tensor və hər bir xüsusiyyətin təsvirini. Təsvir, hər bir xüsusiyyət adını ya **tf.io.FixedLenFeature** descriptor-u ilə əlaqələndirən bir lüğətdir ki, bu da xüsusiyyətin ölçüsünü, növünü və default dəyərini göstərir, ya da **tf.io.VarLenFeature** descriptor-u ilə, əgər xüsusiyyətin siyahısının uzunluğu dəyişkən ola bilərsə (məsələn, "emails" xüsusiyyəti üçün).

Aşağıdakı kod bir təsvir lüğəti təyin edir, sonra bir **TFRecordDataset** yaradır və hər bir seriyalaşdırılmış **Example** protobuf-ı ayırmaq üçün xüsusi bir əvvəlcədən işləmə funksiyasını tətbiq edir:

```
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),
}

def parse(serialized_example):
    return tf.io.parse_single_example(serialized_example, feature_description)

dataset = tf.data.TFRecordDataset(["my_contacts.tfrecord"]).map(parse)

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

Sabit uzunluqlu xüsusiyyətlər adi tensorlar kimi ayrılır, lakin dəyişkən uzunluqlu xüsusiyyətlər sıxışdırılmış tensorlar (sparse tensors) kimi ayrılır. Sıxışdırılmış tensoru sıx olmayan bir tensora çevirmək üçün **tf.sparse.to_dense()** istifadə edə bilərsiniz, amma bu halda sadəcə onun dəyərlərinə müraciət etmək daha sadədir:

```
>>> tf.sparse.to_dense(parsed_example["emails"], default_value=b"")
<tf.Tensor: [...] dtype=string, numpy=array([b'a@b.com', b'c@d.com'], [...])>
>>> parsed_example["emails"].values
<tf.Tensor: [...] dtype=string, numpy=array([b'a@b.com', b'c@d.com'], [...])>
```

tf.io.parse_single_example() ilə nümunələri bir-bir ayrmaq yerinə, onları **tf.io.parse_example()** istifadə edərək toplu şəkildə ayırmaq istəyirsinizsə, aşağıdakı kodu istifadə edə bilərsiniz:

```
def parse(serialized_examples):
    return tf.io.parse_example(serialized_examples, feature_description)

dataset = tf.data.TFRecordDataset(["my_contacts.tfrecord"]).batch(2).map(parse)

for parsed_examples in dataset:
    print(parsed_examples) # iki nümunə bir anda
```

Son olaraq, bir **BytesList** istədiyiniz hər hansı bir binary məlumatı ehtiva edə bilər, o cümlədən hər hansı bir seriyalaşdırılmış obyekt. Məsələn, **tf.io.encode_jpeg()** istifadə edərək bir şəkli JPEG formatında kodlaya bilərsiniz və bu binary məlumatı bir **BytesList**-də saxlaya bilərsiniz. Daha sonra kodunuz **TFRecord**-u oxuduqda, əvvəlcə **Example**-ı ayıracaq, sonra isə məlumatı ayırmaq və orijinal şəkli əldə etmək üçün **tf.io.decode_jpeg()** çağırması etməlidir (ya da hər hansı bir BMP, GIF, JPEG və ya PNG şəkilini ayıra bilən **tf.io.decode_image()** istifadə edə bilərsiniz). Həmçinin, istədiyiniz hər hansı bir tensoru **BytesList**-də saxlaya bilərsiniz, bunu **tf.io.serialize_tensor()** istifadə edərək serializasiya edib, nəticə olan byte zəncirini **BytesList** xüsusiyyətində yerləşdirərək. Daha sonra **TFRecord**-u ayırdığınız zaman bu məlumatı **tf.io.parse_tensor()** istifadə edərək ayıra bilərsiniz. Bu fəsildə şəkilləri və tensorları **TFRecord** faylında saxlamağa dair nümunələri https://homl.info/colab3 ünvanında tapa bilərsiniz.

Göründüyü kimi, **Example** protobuf-u olduqca çevikdir, buna görə də əksər istifadə halları üçün kifayət edəcək. Ancaq listələrin listələri ilə işləyərkən istifadə etmək bir qədər çətin ola bilər. Məsələn, tutaq ki, siz mətn sənədlərini təsnif etmək istəyirsiniz. Hər bir sənəd cümlələrin siyahısı ilə təmsil oluna bilər, burada hər bir cümlə sözlərin siyahısı ilə təmsil olunur. Və bəlkə hər bir sənədin həm də şərhlər siyahısı var, burada hər bir şərh sözlərin siyahısı ilə təmsil olunur. Ola bilsin ki, bəzi kontekstual məlumatlar da olsun, məsələn, sənədin müəllifi, başlığı və nəşr tarixi. TensorFlow-un **SequenceExample** protobuf-u belə istifadə hallarına görə nəzərdə tutulmuşdur.

Siyahıların Siyahıları ilə İşləmək üçün SequenceExample Protobuf-u

SequenceExample protobuf-unun tərifi belədir:

```
message FeatureList {
        repeated Feature feature = 1;
};

message FeatureLists {
        map<string, FeatureList> feature_list = 1;
};

message SequenceExample {
        Features context = 1;
        FeatureLists feature_lists = 2;
};
```

Bir **SequenceExample** kontekstual məlumatlar üçün bir **Features** obyektini və bir və ya bir neçə adlandırılmış **FeatureList** obyektini (məsələn, "content" adlı bir **FeatureList** və "comments" adlı başqa bir **FeatureList**) ehtiva edir. Hər bir **FeatureList Feature** obyektlərinin siyahısını ehtiva edir, hansılar ki, byte zəncirlərinin siyahısı, 64-bit tam ədədlərinin siyahısı və ya float-ların siyahısı ola bilər (bu nümunədə, hər bir **Feature** cümlə və ya şərh təmsil edə bilər, bəlkə də söz identifikatorlarının siyahısı şəklində).

Bir **SequenceExample** qurmaq, onu seriyalaşdırmaq və ayrılmasını həyata keçirmək, **Example** ilə işləmək kimi oxşar olsa da, burada **tf.io.parse_single_sequence_example()** funksiyasından bir tək **SequenceExample** ayırmaq üçün və ya **tf.io.parse_sequence_example()** funksiyasından bir toplu ayırmaq üçün istifadə etməlisiniz. Hər iki funksiya kontekstual xüsusiyyətlər (lüğət şəklində) və xüsusiyyət siyahıları (yəni, başqa bir lüğət şəklində) ehtiva edən bir tuple qaytarır. Əgər xüsusiyyət siyahıları dəyişkən ölçülü ardıcıllıqlar ehtiva edirsə (yuxarıdakı nümunə kimi), onları **tf.RaggedTensor.from_sparse()** istifadə edərək **ragged tensor**-a çevirmək istəyirsinizsə, tam kodu görmək üçün notbuku yoxlaya bilərsiniz:

```
parsed_context, parsed_feature_lists = tf.io.parse_single_sequence_example(
        serialized_sequence_example, context_feature_descriptions,
        sequence_feature_descriptions)
parsed_content = tf.RaggedTensor.from_sparse(parsed_feature_lists["content"])
```

İndi bilirsiniz ki, **tf.data API**, **TFRecords** və **protobuf-lar** istifadə edərək məlumatları səmərəli şəkildə saxlamaq, yükləmək, ayırmaq və əvvəlcədən işləmək mümkündür, növbəti addım **Keras preprocessing layers**-a diqqət yetirməkdir.

Keras Əvvəlcədən İşləmə Qatları

Sinir şəbəkəsi üçün məlumatları hazırlamaq adətən aşağıdakıları tələb edir:

- Sayısal xüsusiyyətlərin normalizasiyası,
- Kateqoriyalı xüsusiyyətlər və mətni kodlaşdırmaq,
- Şəkillərin kəsilməsi və ölçüsünün dəyişdirilməsi və s.

Bu məqsədlə bir neçə seçim mövcuddur:

1. **Əvvəlcədən işləmə** - Təlim məlumat fayllarınızı hazırlayarkən bu əməliyyatları əvvəlcədən edə bilərsiniz. NumPy, Pandas və ya Scikit-Learn kimi istənilən alətlərdən istifadə edə bilərsiniz. Təlim prosesində istifadə etdiyiniz məlumatları düzgün şəkildə eyni preprocessing addımlarını tətbiq edərək istehsalatda da eyni addımları tətbiq etməlisiniz.
2. **Verilənləri yüklərkən preprocessing** - **tf.data** ilə məlumatları yükləyərkən preprocessing əməliyyatını tətbiq edə bilərsiniz, bunun üçün həmin dataset-ə **map()** metodu ilə preprocessing funksiyasını tətbiq etməlisiniz. Yuxarıda göstərdiyimiz kimi. Bu yanaşmada da, istehsalatda eyni preprocessing addımlarını tətbiq etməlisiniz.
3. **Model daxilində preprocessing qatmanları** - Ən son yanaşma, preprocessing qatmanlarını birbaşa modelinizə daxil etməkdir, beləliklə model təlim zamanı bütün daxil olan məlumatları anında əvvəlcədən işləyəcək və eyni preprocessing qatmanlarını istehsalatda istifadə edə biləcəksiniz. Bu fəsil bu son yanaşmanı əhatə edəcək.

Keras, modellərinizə daxil edə biləcəyiniz çoxsaylı preprocessing qatmanları təklif edir: bunlar sayısal xüsusiyyətlərə, kateqoriyalı xüsusiyyətlərə, şəkillərə və mətni işləməyə tətbiq oluna bilər. Növbəti bölmələrdə sayısal və kateqoriyalı xüsusiyyətlərdən, həmçinin sadə mətn preprocessing-dən bəhs edəcəyik, şəkil preprocessing-ə isə 14-cü fəsildə, daha inkişaf etmiş mətn preprocessing-ə isə 16-cı fəsildə nəzər salacağıq.

Normalizasiya Qatı

10-cu fəsildə gördüyümüz kimi, Keras giriş xüsusiyyətlərini standartlaşdırmaq üçün **Normalization** qatmanı təqdim edir. Bu qatmanı yaratarkən hər bir xüsusiyyətin orta və dispersiyasını təyin edə bilərik və ya daha sadə olaraq, modeli uyğunlaşdırmadan əvvəl təlim dəstini qatmanın **adapt()** metoduna verə bilərik ki, qatman öz başına xüsusiyyətlərin ortalarını və dispersiyalarını ölçsün:

```
norm_layer = tf.keras.layers.Normalization()

model = tf.keras.models.Sequential([

        norm_layer,

        tf.keras.layers.Dense(1)

])

model.compile(loss="mse", optimizer=tf.keras.optimizers.SGD(learning_rate=2e-3))

norm_layer.adapt(X_train) # hər bir xüsusiyyətin orta və dispersiyasını hesablayır

model.fit(X_train, y_train, validation_data=(X_valid, y_valid), epochs=5)
```

IPUCU
adapt() metoduna verilən məlumat nümunəsi, verilənlər dəstinizin təmsil edilməsi üçün kifayət qədər böyük olmalıdır, amma tam təlim dəsti olması lazım deyil: Normalization qatmanı üçün təlim dəstindən təsadüfi seçilmiş bir neçə yüz nümunə ümumiyyətlə xüsusiyyətlərin ortaları və dispersiyaları barədə yaxşı bir təxmin əldə etmək üçün kifayətdir.

Bundan sonra **Normalization** qatmanını modelə daxil etdiyimiz üçün, bu modeli istehsalata təqdim edərkən artıq normalizasiya haqqında düşünməyə ehtiyacımız yoxdur: model bunu özü idarə edəcək (Şəkil 13-4-ə baxın). Fantastik! Bu yanaşma, insanlarda təlim və istehsalat üçün fərqli preprocessing kodlarını saxlamağa çalışarkən baş verən preprocessing uyğunsuzluğu riskini tamamilə aradan qaldırır; yəni birini yeniləyib digərlərini unudurlar. İstehsalat modeli sonra gözlənilməyən şəkildə əvvəlcədən işlənmiş məlumat alır. Əgər şanslıdırlar, aydın bir səhv alırlar. Əgər yox, modelin dəqiqliyi sadəcə səssizcə pisləşir.

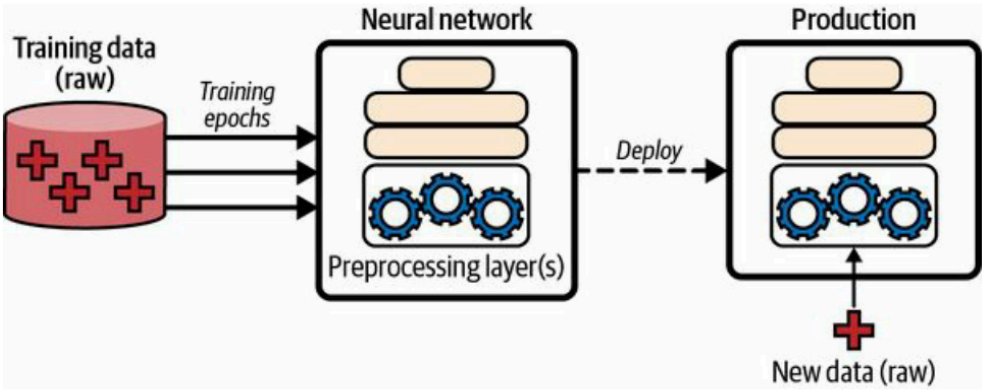

Figure 13-4. Including preprocessing layers inside a model

Modelə preprocessing qatmanını birbaşa daxil etmək yaxşı və sadədir, lakin təlimi bir az (yalnız **Normalization** qatmanı üçün çox az) yavaşlada bilər: əslində, preprocessing təlim zamanı bir dəfə hər epoxda həyata keçirilir. Daha yaxşı nəticə əldə etmək üçün təlimdən əvvəl tam təlim dəstini bir dəfə normalizasiya edə bilərik. Bunu etmək üçün **Normalization** qatmanını ayrıca istifadə edə bilərik (Scikit-Learn **StandardScaler**-a bənzər bir şəkildə):

```
norm_layer = tf.keras.layers.Normalization()

norm_layer.adapt(X_train)

X_train_scaled = norm_layer(X_train)

X_valid_scaled = norm_layer(X_valid)
```

İndi normalizasiya edilmiş məlumatlarla model təlimi apara bilərik, bu dəfə **Normalization** qatmanı olmadan:

```
model = tf.keras.models.Sequential([tf.keras.layers.Dense(1)])

model.compile(loss="mse", optimizer=tf.keras.optimizers.SGD(learning_rate=2e-3))

model.fit(X_train_scaled, y_train, epochs=5, validation_data=(X_valid_scaled, y_valid))
```

Yaxşı! Bu, təlimi bir az sürətləndirməlidir. Lakin indi modelimizi istehsalata təqdim edərkən girişlərini preprocessing etməyəcək. Bunun üçün sadəcə olaraq **Normalization** qatmanını və təlim etdiyimiz modeli əhatə edən yeni bir model yaratmalıyıq. Daha sonra bu son modeli istehsalata təqdim edərək həm girişlərini preprocessing edəcək, həm də proqnozlaşdırma edəcək (Şəkil 13-5-ə baxın):

```
final_model = tf.keras.Sequential([norm_layer, model])

X_new = X_test[:3]  # bir neçə yeni nümunə qəbul edirik (normalizasiya edilməmiş)

y_pred = final_model(X_new)  # məlumatları preprocessing edir və proqnozlar verir
```

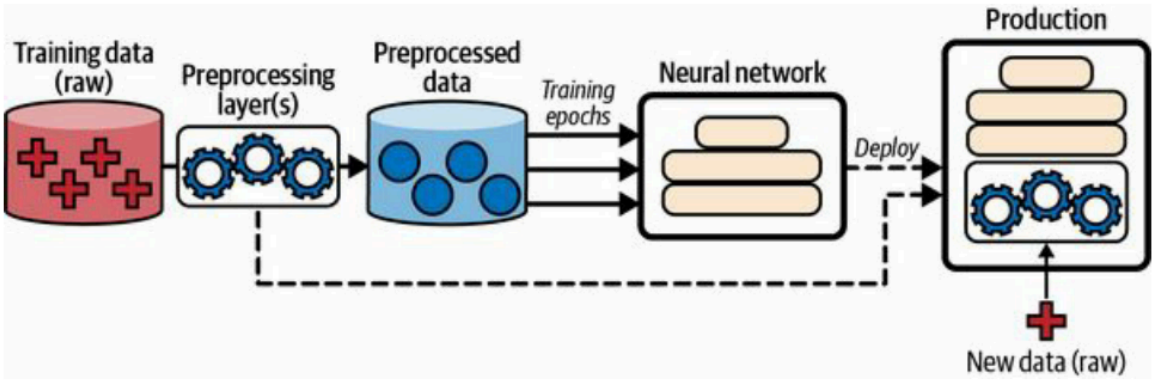

Figure 13-5. Preprocessing the data just once before training using preprocessing layers, then deploying these layers inside the final model

İndi həm sürətli təlimimiz var, çünki yalnız təlimə başlamazdan əvvəl məlumatları bir dəfə preprocessing edirik, həm də son modelimiz girişlərini dinamik şəkildə preprocessing edə bilir, hər hansı bir preprocessing uyumsuzluğu riski olmadan. Üstəlik, Keras preprocessing qatmanları **tf.data API** ilə yaxşı işləyir. Məsələn, bir **tf.data.Dataset** obyektini preprocessing qatmanının **adapt()** metoduna keçirmək mümkündür. Eyni zamanda, bir Keras preprocessing qatmanını **tf.data.Dataset** obyektinə tətbiq etmək üçün datasetin **map()** metodundan istifadə etmək də mümkündür. Məsələn, aşağıda göstərildiyi kimi adaptasiya olunmuş **Normalization** qatmanını datasetdəki hər bir partiyanın giriş xüsusiyyətlərinə tətbiq edə bilərsiniz:

```
dataset = dataset.map(lambda X, y: (norm_layer(X), y))
```

Son olaraq, əgər Keras preprocessing qatmanlarının təmin etdiyi xüsusiyyətlərdən daha çox xüsusiyyətə ehtiyacınız varsa, özünüz də bir Keras qatmanı yaza bilərsiniz, məhz bu barədə 12-ci fəsildə danışmışdıq. Məsələn, əgər **Normalization** qatmanı mövcud olmasaydı, aşağıdakı özəl qatmanı istifadə edərək oxşar nəticə əldə edə bilərdiniz:

```
import numpy as np

class MyNormalization(tf.keras.layers.Layer):

        def adapt(self, X):

        self.mean_ = np.mean(X, axis=0, keepdims=True)

        self.std_ = np.std(X, axis=0, keepdims=True)



        def call(self, inputs):

        eps = tf.keras.backend.epsilon()  # kiçik bir yumşaltma termi

        return (inputs - self.mean_) / (self.std_ + eps)
```

Növbəti addımda, növbəti Keras preprocessing qatmanlarından biri olan **Discretization** qatmanına baxaq.

Diskretləşdirmə Qatmanı

Diskretləşdirmə qatmanının məqsədi, rəqəmsal xüsusiyyəti kateqoriyaya çevirməkdir, bunun üçün dəyər aralıqları (binlər adlanır) kateqoriyalara uyğunlaşdırılır. Bu, bəzən çoxmodal paylanmaları olan və ya hədəf ilə yüksək qeyri-xətti əlaqəyə malik xüsusiyyətlər üçün faydalıdır. Məsələn, aşağıdakı kod rəqəmsal yaş xüsusiyyətini üç kateqoriyaya xəritələyir: 18-dən kiçik, 18-50 (daxil edilmir) və 50 və ya daha yuxarı:

```
>>> age = tf.constant([[10.], [93.], [57.], [18.], [37.], [5.]])

>>> discretize_layer = tf.keras.layers.Discretization(bin_boundaries=[18., 50.])

>>> age_categories = discretize_layer(age)

>>> age_categories

<tf.Tensor: shape=(6, 1), dtype=int64, numpy=array([[0],[2],[2],[1],[1],[0]])>
```

Bu nümunədə, biz tələb olunan bin sərhədlərini təmin etdik. İstəsəniz, bin sayı (num_bins) verə və qatmanın **adapt()** metodunu çağıraraq, uyğun bin sərhədlərini tapmasına icazə verə bilərsiniz. Məsələn, **num_bins=3** təyin etsək, bin sərhədləri 33-cü və 66-cı faizilərin (bu nümunədə 10 və 37) altındakı dəyərlərdə yerləşəcək:

```
>>> discretize_layer = tf.keras.layers.Discretization(num_bins=3)

>>> discretize_layer.adapt(age)

>>> age_categories = discretize_layer(age)

>>> age_categories

<tf.Tensor: shape=(6, 1), dtype=int64, numpy=array([[1],[2],[2],[1],[2],[0]])>
```

Bu cür kateqoriya identifikatorları adətən birbaşa neyron şəbəkəsinə verilməməlidir, çünki onların dəyərləri mənalı şəkildə müqayisə edilə bilməz. Bunun əvəzinə, onlar kodlanmalıdır, məsələn, **one-hot encoding** istifadə edərək. Gəlin, bunu necə edəcəyimizi indi nəzərdən keçirək.

Kateqoriya Kodlaşdırma Qatmanı

Əgər yalnız bir neçə kateqoriya varsa (məsələn, bir-iki onluqdan az), o zaman **one-hot encoding** tez-tez yaxşı bir seçimdir (2-ci fəsildə müzakirə edildiyi kimi). Bunu etmək üçün Keras **CategoryEncoding** qatmanını təqdim edir. Məsələn, yaradılmış **age_categories** xüsusiyyətini **one-hot encoding** ilə kodlayaq:

```
>>> onehot_layer = tf.keras.layers.CategoryEncoding(num_tokens=3)

>>> onehot_layer(age_categories)

<tf.Tensor: shape=(6, 3), dtype=float32, numpy=

array([[0., 1., 0.],

[0., 0., 1.],
```

[0., 0., 1.],

[0., 1., 0.],

[0., 0., 1.],

[1., 0., 0.]], dtype=float32)>

Əgər eyni anda bir neçə kateqorik xüsusiyyəti kodlamağa çalışsanız (bu yalnız onlar eyni kateqoriyaları istifadə edərsə məna kəsb edir), **CategoryEncoding** sinifi standart olaraq **multi-hot encoding** tətbiq edəcək: çıxış tensoru, hər bir giriş xüsusiyyətində mövcud olan kateqoriya üçün 1 olacaq. Məsələn:

>>> two_age_categories = np.array([[1, 0], [2, 2], [2, 0]])

>>> onehot_layer(two_age_categories)

<tf.Tensor: shape=(3, 3), dtype=float32, numpy=

array([[1., 1., 0.],

[0., 0., 1.],

[1., 0., 1.]], dtype=float32)>

Əgər hər bir kateqoriyanın neçə dəfə baş verdiyini bilmək faydalı olarsa, **CategoryEncoding** qatmanını yaradarkən **output_mode="count"** təyin edə bilərsiniz, bu halda çıxış tensoru hər bir kateqoriyanın neçə dəfə təkrarlandığını göstərəcək. Yuxarıdakı nümunədə, çıxış eyni olacaq, yalnız ikinci sətir [0., 0., 2.] olaraq dəyişəcək.

Qeyd edək ki, həm **multi-hot encoding**, həm də **count encoding** məlumat itkisinə səbəb olur, çünki aktiv olan hər bir kateqoriyanın hansı xüsusiyyətdən gəldiyini bilmək mümkün deyil. Məsələn, həm [0, 1], həm də [1, 0] [1., 1., 0.] olaraq kodlanır. Bunu qarşısını almaq istəyirsinizsə, hər bir xüsusiyyəti ayrıca **one-hot encoding** ilə kodlaya və nəticələri birləşdirə bilərsiniz. Bu halda, [0, 1] [1., 0., 0., 0., 1., 0.] olaraq, [1, 0] isə [0., 1., 0., 1., 0., 0.] olaraq kodlanacaq. Eyni nəticəni, kateqoriya identifikatorlarını dəyişdirərək əldə edə bilərsiniz ki, onlar bir-biri ilə üst-üstə düşməsin. Məsələn:

>>> onehot_layer = tf.keras.layers.CategoryEncoding(num_tokens=3 + 3)

>>> onehot_layer(two_age_categories + [0, 3]) # ikinci xüsusiyyətə 3 əlavə edir

<tf.Tensor: shape=(3, 6), dtype=float32, numpy=

array([[0., 1., 0., 1., 0., 0.],

[0., 0., 1., 0., 0., 1.],

[0., 0., 1., 1., 0., 0.]], dtype=float32)>

Bu çıxışda, ilk üç sütun birinci xüsusiyyətə, sonrakı üç sütun isə ikinci xüsusiyyətə uyğundur. Bu, modelə iki xüsusiyyəti ayırmağa imkan verir. Lakin bu, modelə daxil olan xüsusiyyətlərin sayını artırır və beləliklə daha çox model parametrinə ehtiyac yaradır. Hansı kodlaşdırma üsulunun (multi-hot və ya hər bir xüsusiyyət üçün ayrı-ayrı one-hot) ən yaxşı işləyəcəyini əvvəlcədən bilmək çətindir: bu, tapşırıqdan asılıdır və hər iki variantı sınaqdan keçirmək lazım ola bilər.

İndi isə kateqorik tam ədəd xüsusiyyətləri **one-hot** və ya **multi-hot encoding** istifadə edərək kodlaya bilərsiniz. Amma ya kateqorik mətn xüsusiyyətləri necə olacaq? Bunun üçün **StringLookup** qatmanını istifadə edə bilərsiniz.

StringLookup Layihəsi

Gəlin, Keras StringLookup layihəsindən istifadə edərək şəhərlər xüsusiyyətini bir-hot kodlayaq:

```
>>> cities = ["Auckland", "Paris", "Paris", "San Francisco"]

>>> str_lookup_layer = tf.keras.layers.StringLookup()

>>> str_lookup_layer.adapt(cities)

>>> str_lookup_layer([["Paris"], ["Auckland"], ["Auckland"], ["Montreal"]])

<tf.Tensor: shape=(4, 1), dtype=int64, numpy=array([[1], [3], [3], [0]])>
```

Əvvəlcə bir StringLookup layihəsi yaradırıq, sonra onu verilənlərə uyğunlaşdırırıq: bu, üç fərqli kateqoriya tapır. Sonra, layihəni bir neçə şəhəri kodlamaq üçün istifadə edirik. Bunlar, standart olaraq tam ədədlərlə kodlanır. Tanınmayan kateqoriyalar 0 ilə göstərilir, bu misalda "Montreal" kimi. Tanınmış kateqoriyalar ən çox yayılmış kateqoriyadan ən az yayılmışına qədər nömrələnir.

Əlverişli olaraq, StringLookup layihəsini yaratarkən `output_mode="one_hot"` təyin etsəniz, o zaman hər bir kateqoriya üçün bir-hot vektoru çıxaracaq, tam ədəd əvəzinə:

```
>>> str_lookup_layer = tf.keras.layers.StringLookup(output_mode="one_hot")

>>> str_lookup_layer.adapt(cities)

>>> str_lookup_layer([["Paris"], ["Auckland"], ["Auckland"], ["Montreal"]])

<tf.Tensor: shape=(4, 4), dtype=float32, numpy=

array([[0., 1., 0., 0.],
```

[0., 0., 0., 1.],

[0., 0., 0., 1.],

[1., 0., 0., 0.]], dtype=float32)>

İPUCU
Keras həmçinin tamamilə StringLookup layihəsi kimi işləyən IntegerLookup layihəsini təqdim edir, lakin bu, mətnlər deyil, tam ədədlər qəbul edir.

Əgər təlim dəsti çox böyükdürsə, layihəni yalnız təlim dəstinin təsadüfi bir alt dəstinə uyğunlaşdırmaq faydalı ola bilər. Bu halda, layihənin `adapt()` metodu bəzi nadir kateqoriyaları əldən verə bilər. Standart olaraq, onlar hamısı 0-a xəritələnəcək və model tərəfindən fərqlənməyəcək. Bu riski azaltmaq (eyni zamanda yalnız təlim dəstinin bir alt dəstinə uyğunlaşdırmaq) üçün `num_oov_indices`-i 1-dən böyük bir tam ədədə təyin edə bilərsiniz. Bu, istifadə olunmayan (OOV) qovşaq sayıdır: hər tanımadıq kateqoriya OOV qovşaqlarından birinə təyin ediləcək, həş funksiyası ilə OOV qovşaq sayına görə. Bu, modelin ən azından bəzi nadir kateqoriyaları fərqləndirməsinə imkan verəcək. Məsələn:

```
>>> str_lookup_layer = tf.keras.layers.StringLookup(num_oov_indices=5)

>>> str_lookup_layer.adapt(cities)

>>> str_lookup_layer([["Paris"], ["Auckland"], ["Foo"], ["Bar"], ["Baz"]])

<tf.Tensor: shape=(4, 1), dtype=int64, numpy=array([[5], [7], [4], [3], [4]])>
```

Çünki beş OOV qovşağı var, ilk tanınmış kateqoriyanın ID-si artıq 5-dir ("Paris"). Lakin "Foo", "Bar" və "Baz" tanınmır, ona görə də hər biri OOV qovşaqlarından birinə təyin edilir. "Bar" özünə məxsus bir qovşaq alır (ID 3 ilə), amma təəssüf ki, "Foo" və "Baz" eyni qovşağa təyin olunur (ID 4 ilə), buna görə də onlar model tərəfindən fərqlənməz. Bu, həş çarpışması adlanır. Çarpışma riskini azaltmağın yeganə yolu OOV qovşaqlarının sayını artırmaqdır. Ancaq bu, kateqoriyaların ümumi sayını artıracaq və bir-hot kodlaşdırıldıqda daha çox RAM və əlavə model parametrləri tələb edəcək. Bu səbəbdən bu rəqəmi çox artırmamalısınız.

Bu kateqoriyaları təsadüfi şəkildə qovşaqlara xəritələndirmək ideyasına "həşləmə hiyləsi" deyilir. Keras bunun üçün xüsusi bir qat təqdim edir: Hashing qatı.

Hashing Layihəsi

Hər bir kateqoriya üçün Keras Hashing layihəsi, qovşaq (və ya "bin") sayına görə həş hesablayır. Xəritələmə tamamilə təsadüfi olsa da, işləmələr və platformalar arasında sabit qalır (yəni, eyni kateqoriya həmişə eyni tam ədədə təyin olunur, qovşaq sayına dəyişiklik olmadığı müddətcə). Məsələn, gəlin Hashing layihəsini bir neçə şəhəri kodlamaq üçün istifadə edək:

```
>>> hashing_layer = tf.keras.layers.Hashing(num_bins=10)

>>> hashing_layer([["Paris"], ["Tokyo"], ["Auckland"], ["Montreal"]])

<tf.Tensor: shape=(4, 1), dtype=int64, numpy=array([[0], [1], [9], [1]])>
```

Bu qatın faydası budur ki, heç bir uyğunlaşdırma etməyə ehtiyac yoxdur, bu bəzən faydalı ola bilər, xüsusən də məlumat dəsti yaddaşa sığmayacaq qədər böyük olduqda. Lakin yenə də həş çarpışması ilə qarşılaşırıq: "Tokyo" və "Montreal" eyni ID-yə təyin edilir, bu da onları model tərəfindən fərqlənməz edir. Beləliklə, adətən StringLookup qatına üstünlük vermək yaxşıdır.

İndi kateqoriyaları kodlamağın başqa bir yoluna baxaq: təlim edilə bilən vektorlar (embeddings).

Kateqoriyalı Xüsusiyyətləri Embedding-lərlə Kodlaşdırma

Embedding, daha yüksək ölçülü məlumatların sıx (dense) təmsilidir, məsələn, bir kateqoriya və ya bir söz. Əgər 50,000 mümkün kateqoriya varsa, bir-hot kodlaşdırma 50,000 ölçülü seyrək vektor (yəni, əksər hallarda sıfırlar olan) yaradar. Bunun əksinə olaraq, embedding nisbətən kiçik sıx vektor olacaq; məsələn, yalnız 100 ölçüdə.

Dərin öyrənmədə, embedding-lər adətən təsadüfi şəkildə başlanğıc verilərək, sonra digər model parametrləri ilə birlikdə gradient enişi ilə təlim edilir. Məsələn, Kaliforniya mənzil verilənlər bazasında "NEAR BAY" kateqoriyası əvvəlcə [0.131, 0.890] kimi təsadüfi bir vektorla təmsil oluna bilər, "NEAR OCEAN" kateqoriyası isə başqa bir təsadüfi vektorla, məsələn [0.631, 0.791], təmsil edilə bilər. Bu misalda, 2D embedding istifadə edirik, amma ölçü sayını dəyişə biləcəyiniz bir hiperparametrdir.

Bu embedding-lər təlim edilə bilən olduğundan, təlim zamanı tədricən yaxşılaşacaq; və bu halda, onlar olduqca oxşar kateqoriyaları təmsil etdikləri üçün gradient enişi onları bir-birinə yaxınlaşdıracaq, eyni zamanda "INLAND" kateqoriyasının embedding-indən uzaqlaşdırmağa meylli olacaq (Şəkil 13-6-ya baxın). Həqiqətən də, təmsilin nə qədər yaxşı olarsa, neyron şəbəkəsi üçün dəqiq proqnozlar vermək daha asan olacaq, buna görə də təlim, embedding-ləri kateqoriyaların faydalı təmsillərinə çevirməyə meyllidir. Bu, təmsil öyrənməsi adlanır (bu cür təmsil öyrənmələrinin digər növlərini 17-ci fəsildə görəcəksiniz).

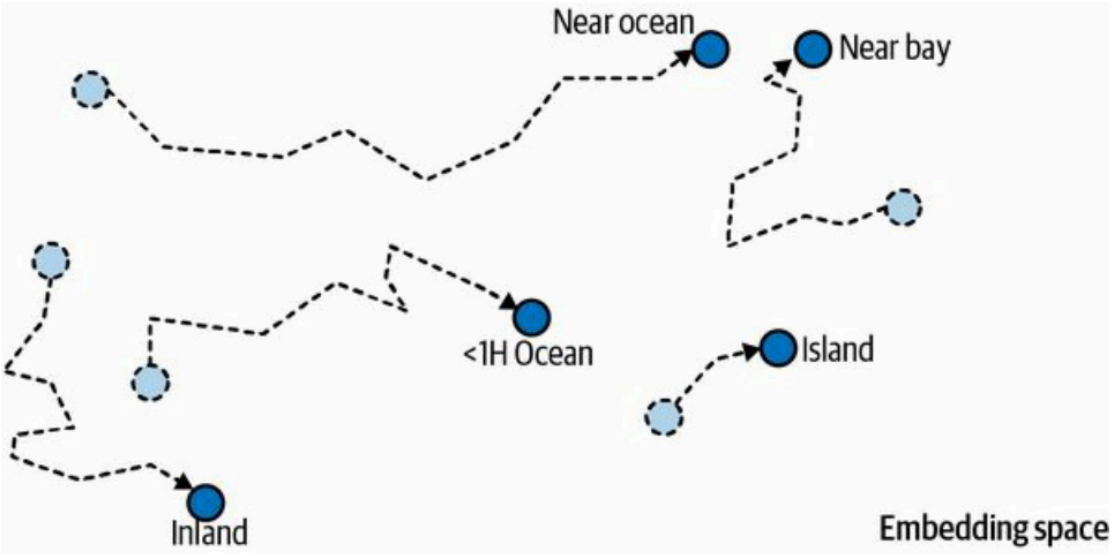

Figure 13-6. Embeddings will gradually improve during training

SÖZ EMBEDDING-ləri

Embedding-lər yalnız mövcud tapşırıq üçün faydalı təmsillər olmaqla qalmır, həm də çox vaxt eyni embedding-lər digər tapşırıqlar üçün uğurla təkrar istifadə oluna bilər. Bunun ən yaygın nümunəsi söz embedding-ləridir (yəni, fərdi sözlərin embedding-ləri): təbii dil emalı tapşırığı üzərində işləyərkən, çox vaxt öz embedding-lərinizi təlim etməkdən daha yaxşı nəticə verən əvvəlcədən təlim edilmiş söz embedding-lərindən istifadə etməkdir.

Sözləri təmsil etmək üçün vektorlar istifadə etmə ideyası 1960-cı illərə qədər gedib çıxır və faydalı vektorlar yaratmaq üçün çoxsaylı mürəkkəb texnikalar istifadə edilib, bunlara neyron şəbəkələrinin istifadəsi də daxildir. Lakin hər şey 2013-cü ildə həqiqətən sıçradı, Tomáš Mikolov və digər Google tədqiqatçıları neyron şəbəkələrdən istifadə edərək söz embedding-lərini öyrənmək üçün effektiv bir texnikanı təsvir edən məqalə nəşr etdilər və əvvəlki cəhdləri əhəmiyyətli dərəcədə geridə qoydular. Bu, onlara çox böyük mətn korpusunda embedding-ləri öyrənməyə imkan verdi: onlar neyron şəbəkəsini verilən hər hansı bir sözün yaxınındakı sözləri proqnozlaşdırmağa öyrətdilər və heyrətamiz söz embedding-ləri əldə etdilər. Məsələn, sinonimlər çox yaxın embedding-lərə sahib oldular, Fransızca, İspaniya və İtaliya kimi semantik cəhətdən əlaqəli sözlər isə bir araya toplandı.

Lakin məsələ yalnız yaxınlıqla bitmir: söz embedding-ləri həmçinin embedding məkanında mənalı oxlar boyunca təşkil edilmişdir. Budur məşhur bir nümunə: əgər King – Man + Woman (bu sözlərin embedding vektorlarını əlavə edib çıxardıqda) hesablasanız, nəticə Queen sözünün embedding-ə çox yaxın olacaq (Şəkil 13-7-ə baxın). Başqa sözlə, söz embedding-ləri gender anlayışını kodlayır!

Eynilə, Madrid – Spain + France hesablayaraq nəticə Parisə çox yaxın olacaq, bu da göstərir ki, paytaxt şəhər anlayışı da embedding-lərdə kodlanıb.

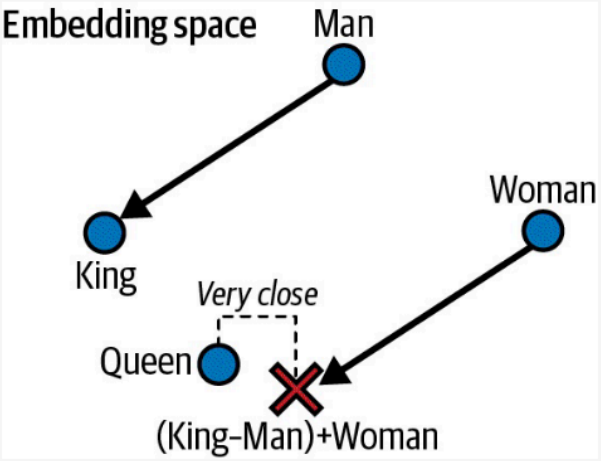

Figure 13-7. Word embeddings of similar words tend to be close, and some axes seem to encode meaningful concepts

Təəssüf ki, söz embedding-ləri bəzən bizim ən pis qərəzlərimizi ələ keçirir. Məsələn, onlar düzgün şəkildə öyrənirlər ki, "Man" sözünün "King"ə bənzərliyi, "Woman" sözünün isə "Queen"ə bənzərliyini, lakin onlar həmçinin öyrənirlər ki, "Man" sözünün "Doctor" ilə əlaqəsi, "Woman"ın isə "Nurse" ilə əlaqəsi var: bu, əslində, çox cinsi qərəzli bir yanaşmadır! Ədalətli olmaq üçün, bu xüsusi nümunə yəqin ki, şişirdilib, çünki 2019-cu ildə Malvina Nissim və digər tədqiqatçılar tərəfindən aparılan bir araşdırmada vurğulandı. Bununla belə, dərin öyrənmə alqoritmlərində ədalətin təmin edilməsi mühüm və aktiv bir araşdırma mövzusudur.

Keras, bir embedding matrisi ilə sarılmış bir Embedding qatını təmin edir: bu matrisi hər bir kateqoriya üçün bir sətir və hər bir embedding ölçüsü üçün bir sütun ehtiva edir. Varsayılan olaraq, bu, təsadüfi şəkildə ilkinləşdirilir. Bir kateqoriya ID-sini embedding-ə çevirmək üçün Embedding qatının özünün sadəcə olaraq o kateqoriyaya uyğun olan sətiri tapması və qaytarması lazımdır. Bu qədər sadədir! Məsələn, gəlin beş sətir və 2D embedding-lərlə bir Embedding qatını ilkinləşdirək və onu bəzi kateqoriyaları kodlamaq üçün istifadə edək:

```
>>> tf.random.set_seed(42)

>>> embedding_layer = tf.keras.layers.Embedding(input_dim=5, output_dim=2)

>>> embedding_layer(np.array([2, 4, 2]))

<tf.Tensor: shape=(3, 2), dtype=float32, numpy=

array([[-0.04663396, 0.01846724],

[-0.02736737, -0.02768031],

[-0.04663396, 0.01846724]], dtype=float32)>
```

Gördüyünüz kimi, kateqoriya 2 (iki dəfə) [-0.04663396, 0.01846724] 2D vektoru ilə kodlanır, 4-cü kateqoriya isə [-0.02736737, -0.02768031] ilə kodlanır. Çünki qat hələ təlim edilməyib, bu kodlamalar sadəcə təsadüfidir.

XƏBƏRDARLIQ Bir Embedding qatı təsadüfi şəkildə ilkinləşdirilir, buna görə də onu modeldən kənarda müstəqil bir preprocessing qat olaraq istifadə etmək mənasızdır, əgər onu əvvəlcədən təlim keçmiş çəki ilə ilkinləşdirmirsinizsə.

Əgər kateqorik mətn atributunu embedding etmək istəyirsinizsə, sadəcə bir **StringLookup** qatı ilə birləşdirə bilərsiniz, beləliklə:

```
>>> tf.random.set_seed(42)

>>> ocean_prox = ["<1H OCEAN", "INLAND", "NEAR OCEAN", "NEAR BAY", "ISLAND"]

>>> str_lookup_layer = tf.keras.layers.StringLookup()

>>> str_lookup_layer.adapt(ocean_prox)

>>> lookup_and_embed = tf.keras.Sequential([

... str_lookup_layer,

... tf.keras.layers.Embedding(input_dim=str_lookup_layer.vocabulary_size(),

... output_dim=2)

... ])

...

>>> lookup_and_embed(np.array([["<1H OCEAN"], ["ISLAND"], ["<1H OCEAN"]]))

<tf.Tensor: shape=(3, 2), dtype=float32, numpy=

array([[-0.01896119, 0.02223358],

[ 0.02401174, 0.03724445],

[-0.01896119, 0.02223358]], dtype=float32)>
```

Qeyd edək ki, embedding matrisi içindəki sətirlərin sayı sözlük ölçüsünə uyğun olmalıdır: bu, tanınan kateqoriyalarla yanaşı, OOV (out-of-vocabulary) çantaları da daxil olmaqla ümumi kateqoriya sayıdır (varsayılan olaraq yalnız biri var). **StringLookup** sinifinin **vocabulary_size()** metodu bu rəqəmi əlverişli şəkildə qaytarır.

İPUCU Bu nümunədə 2D embedding-lərdən istifadə etdik, amma ümumiyyətlə embedding-lər adətən 10 ilə 300 ölçüsü arasında olur, bu da tapşırıq, sözlük ölçüsü və təlim dəstinizin ölçüsündən asılıdır. Bu hiperparametri tənzimləməlisiniz.

İndi hər şeyi bir araya gətirərək, bir Keras modeli yarada bilərik ki, bu model kateqorik mətn xüsusiyyətlərini və adi sayısal xüsusiyyətləri işləyə bilsin və hər bir kateqoriya (həmçinin hər bir OOV çantasını) üçün embedding öyrənsin:

```
X_train_num, X_train_cat, y_train = [...]  # təlim dəstini yükləyin

X_valid_num, X_valid_cat, y_valid = [...]  # və doğrulama dəstini


num_input = tf.keras.layers.Input(shape=[8], name="num")

cat_input = tf.keras.layers.Input(shape=[], dtype=tf.string, name="cat")

cat_embeddings = lookup_and_embed(cat_input)

encoded_inputs = tf.keras.layers.concatenate([num_input, cat_embeddings])

outputs = tf.keras.layers.Dense(1)(encoded_inputs)

model = tf.keras.models.Model(inputs=[num_input, cat_input], outputs=[outputs])

model.compile(loss="mse", optimizer="sgd")

history = model.fit((X_train_num, X_train_cat), y_train, epochs=5,

validation_data=((X_valid_num, X_valid_cat), y_valid))
```

Bu model iki giriş qəbul edir: **num_input**, hər bir nümunə üçün səkkiz sayısal xüsusiyyətləri ehtiva edir, və **cat_input**, hər bir nümunə üçün bir kateqorik mətn girişi ehtiva edir. Model, əvvəl yaratdığımız **lookup_and_embed** modelindən istifadə edərək hər bir okean yaxınlığı kateqoriyasını müvafiq öyrənilən embedding olaraq kodlayır. Sonra, sayısal girişləri və embedding-ləri **concatenate()** funksiyası ilə birləşdirir və tam kodlanmış girişlər hazırlayır, hansı ki neyron şəbəkəyə verilməyə hazırdır. Bu nöqtədə hər hansı bir neyron şəbəkə əlavə edə bilərik, amma sadəlik üçün sadəcə bir sıx çıxış qatı əlavə edirik, sonra isə Keras modelini yaratdıqdan sonra onu tərtib edib təlim edirik, həm sayısal, həm də kateqorik girişləri keçiririk.

QEYD : One-hot encoding-in ardından Dense qatının istifadə edilməsi (aktivasiya funksiyası və ya yanlılıq olmadan) bir Embedding qatına bərabərdir. Lakin, Embedding qatının istifadəsi daha az hesablamalar tələb edir, çünki sıfırla çoxlu sayda vurulmağı qarşısını alır—performans fərqi, embedding matrisinin ölçüsü böyüdükcə aydın olur. Dense qatının çəki matrisi, embedding matrisinin rolunu oynayır. Məsələn, 20 ölçülü one-hot vektorları və 10 vahidli bir Dense qatının istifadəsi, input_dim=20 və output_dim=10 olan bir Embedding qatının istifadəsinə bərabərdir. Nəticədə, embedding qatının izləyən qatında olan vahidlərin sayından daha çox embedding ölçüsü istifadə etmək israf olardı.

İndi ki, kateqorik xüsusiyyətlərin kodlanmasını öyrəndiniz, diqqətimizi mətnin preprocessing-ə yönəltməyin vaxtıdır.

Mətnin əvvəlcədən emalı

Keras, əsas mətnin əvvəlcədən emalı üçün **TextVectorization** təbəqəsini təqdim edir. **StringLookup** təbəqəsinə bənzər şəkildə, ya təbəqənin yaradılması zamanı bir lüğət ötürməli, ya da təlim məlumatlarından istifadə edərək lüğəti öyrənməsi üçün **adapt()** metodunu çağırmalısınız. Gəlin bir nümunəyə baxaq:

```
>>> train_data = ["To be", "!(to be)", "That's the question", "Be, be, be."]

>>> text_vec_layer = tf.keras.layers.TextVectorization()

>>> text_vec_layer.adapt(train_data)

>>> text_vec_layer(["Be good!", "Question: be or be?"])

<tf.Tensor: shape=(2, 4), dtype=int64, numpy=

array([[2, 1, 0, 0],

       [6, 2, 1, 2]])>
```

"Be good!" və "Question: be or be?" cümlələri uyğun olaraq [2, 1, 0, 0] və [6, 2, 1, 2] kimi kodlaşdırılıb. Lüğət təlim məlumatında olan dörd cümlədən öyrənilmişdir: “be” = 2, “to” = 3 və s. Lüğəti qurmaq üçün **adapt()** metodu əvvəlcə təlim cümlələrini kiçik hərflərə çevirmiş və durğu işarələrini silmişdir. Buna görə “Be”, “be” və “be?” hamısı “be” = 2 kimi kodlaşdırılıb. Daha sonra cümlələr boşluqlara əsasən ayrılmış və yaranan sözlər azalan tezliklə sıralanmışdır ki, bu da son lüğəti yaratmışdır. Cümlələrin kodlaşdırılmasında naməlum sözlər 1 kimi kodlaşdırılır.

Nəhayət, birinci cümlə ikinciyə nisbətən daha qısa olduğu üçün 0-larla tamamlanmışdır.

İPUCU
TextVectorization təbəqəsinin bir çox seçimləri var. Məsələn, standartizasiya üçün `standardize=None` təyin edərək hərf ölçüsünü və durğu işarələrini saxlaya bilərsiniz və ya öz standartizasiya funksiyanızı keçə bilərsiniz. Ayrılmanı dayandırmaq üçün `split=None` təyin edə və ya öz ayırma funksiyanızı verə bilərsiniz. Nəticə ardıcıllığının hamısının müəyyən uzunluğa kəsilməsi və ya tamamlanmasını təmin etmək üçün `output_sequence_length` arqumentini təyin edə bilərsiniz və ya adi tensor əvəzinə **ragged** tensor əldə etmək üçün `ragged=True` təyin edə bilərsiniz. Daha çox seçim üçün sənədlərə baxın.

Söz ID-ləri adətən **Embedding** təbəqəsindən istifadə edərək kodlaşdırılmalıdır: bunu 16-cı fəsildə edəcəyik. Alternativ olaraq, **TextVectorization** təbəqəsinin `output_mode` arqumentini "multi_hot" və ya "count" kimi təyin edə bilərsiniz. Ancaq sadəcə sözləri saymaq adətən ideal deyil: məsələn, “to” və “the” kimi sözlər çox yayğındır və çox az əhəmiyyət

daşıyır, halbuki “basketball” kimi nadir sözlər daha məlumatlıdır. Buna görə, "multi_hot" və ya "count" əvəzinə, "tf_idf" seçmək daha yaxşıdır.

TF-IDF, yəni *term-frequency* × *inverse-document-frequency* deməkdir. Bu metodda tez-tez rast gəlinən sözlər azaldılır, nadir sözlər isə gücləndirilir.

```
>>> text_vec_layer = tf.keras.layers.TextVectorization(output_mode="tf_idf")

>>> text_vec_layer.adapt(train_data)

>>> text_vec_layer(["Be good!", "Question: be or be?"])

<tf.Tensor: shape=(2, 6), dtype=float32, numpy=

array([[0.96725637, 0.6931472 , 0. , 0. , 0. , 0. ],

       [0.96725637, 1.3862944 , 0. , 0. , 0. , 1.0986123 ]], dtype=float32)>
```

TF-IDF metodunun bir çox variantı mövcuddur, lakin **TextVectorization** təbəqəsi onu aşağıdakı şəkildə tətbiq edir: hər bir sözün sayı **log(1 + d / (f + 1))** ilə vurulur. Burada **d** təlim məlumatlarındakı ümumi cümlələrin (digər adı ilə sənədlərin) sayını, **f** isə həmin cümlələrdən neçəsinin verilmiş sözü ehtiva etdiyini göstərir. Məsələn, bu halda **d = 4** cümlə var və "be" sözü bunlardan **f = 3**-də mövcuddur. "Question: be or be?" cümləsində "be" iki dəfə təkrarlandığı üçün onun kodlaşdırması **2 × log(1 + 4 / (1 + 3)) ≈ 1.3862944** olur. "question" sözü isə yalnız bir dəfə görünür, lakin daha az rast gəlinən bir söz olduğu üçün onun kodlaşdırması demək olar ki, eyni yüksəkdir: **1 × log(1 + 4 / (1 + 1)) ≈ 1.0986123**. Naməlum sözlər üçün isə orta çəki istifadə olunur.

Bu mətn kodlaşdırma yanaşması istifadəsi asandır və əsas təbii dil işlənməsi tapşırıqları üçün kifayət qədər yaxşı nəticələr verə bilər, lakin bir neçə əhəmiyyətli məhdudiyyətləri var: yalnız sözləri boşluqlarla ayıran dillər üçün işləyir, homonimləri fərqləndirmir (məsələn, “to bear” və “teddy bear”), “evolution” və “evolutionary” kimi sözlərin əlaqəli olduğunu modelə göstərmir və s. Həmçinin, multi-hot, count və ya TF-IDF kodlaşdırmadan istifadə edildikdə sözlərin ardıcıllığı itirilir. Bəs digər seçimlər hansılardır?

Bir seçim **TensorFlow Text** kitabxanasından istifadə etməkdir. Bu kitabxana **TextVectorization** təbəqəsindən daha inkişaf etmiş mətnin əvvəlcədən emalı xüsusiyyətləri təklif edir. Məsələn, mətnin sözlərdən daha kiçik vahidlərə (tokenlərə) bölünməsini təmin edən bir neçə alt sözdizim tokenizatoru mövcuddur. Bu, modelə "evolution" və "evolutionary" kimi sözlər arasında ümumi bir əlaqə olduğunu daha asanlıqla aşkar etməyə imkan verir (alt sözdizim tokenizasiyası haqqında 16-cı fəsildə daha ətraflı danışılacaq).

Başqa bir seçim isə əvvəlcədən təlim keçmiş dil modeli komponentlərindən istifadə etməkdir.

Əvvəlcədən Təlim Keçmiş Dil Modeli Komponentlərindən İstifadə

TensorFlow Hub kitabxanası əvvəlcədən təlim keçmiş model komponentlərini öz modellərinizdə istifadəyə asanlıqla daxil etməyə imkan verir. Bu komponentlərə modul deyilir.

Sadəcə **TF Hub** anbarını nəzərdən keçirin, ehtiyacınız olan modulu tapın və verilmiş kod nümunəsini layihənizə daxil edin. Modul avtomatik olaraq endiriləcək və Keras təbəqəsinə birləşdiriləcək ki, onu birbaşa modelinizdə istifadə edə biləsiniz. Modullar adətən həm əvvəlcədən emal kodunu, həm də əvvəlcədən təlim keçmiş çəkiləri ehtiva edir və ümumiyyətlə əlavə təlim tələb etmir (lakin, əlbəttə, modelinizin qalan hissəsi təlim tələb edəcək).

Məsələn, bəzi güclü əvvəlcədən təlim keçmiş dil modelləri mövcuddur. Ən güclüləri kifayət qədər böyükdür (bir neçə gigabayt), lakin tez bir nümunə üçün **nnlm-en-dim50** modulundan, versiya 2-dən istifadə edək. Bu, xam mətni giriş olaraq qəbul edən və 50 ölçülü cümlə təsvirləri (embedding) yaradan sadə bir moduldur.

```
>>> import tensorflow_hub as hub

>>> hub_layer = hub.KerasLayer("https://tfhub.dev/google/nnlm-en-dim50/2")

>>> sentence_embeddings = hub_layer(tf.constant(["To be", "Not to be"]))

>>> sentence_embeddings.numpy().round(2)

array([[-0.25, 0.28, 0.01, 0.1 , [...] , 0.05, 0.31],

       [-0.2 , 0.2 , -0.08, 0.02, [...] , -0.04, 0.15]], dtype=float32)
```

hub.KerasLayer təbəqəsi modulu verilmiş URL-dən yükləyir. Bu konkret modul cümlə kodlaşdırıcısıdır: giriş olaraq mətnləri qəbul edir və hər birini tək bir vektora (bu halda, 50 ölçülü bir vektora) çevirir. Daxildə, mətnləri ayrır (sözləri boşluqlara əsasən bölür) və hər sözü Google News 7B korpusunda (yeddi milyard sözlük) əvvəlcədən təlim keçmiş embedding matrisindən istifadə edərək kodlaşdırır. Daha sonra bütün söz kodlaşdırmalarının ortalamasını hesablayır və nəticədə cümlə təsviri alınır.

Sadəcə bu **hub_layer** təbəqəsini modelinizə daxil edin və istifadəyə hazırsınız. Qeyd edək ki, bu konkret dil modeli İngilis dili üçün təlim keçib, lakin digər dillər, eləcə də çoxdilli modellər də mövcuddur.

Son olaraq, **Hugging Face** tərəfindən hazırlanmış açıq mənbəli **Transformers** kitabxanası da güclü dil modeli komponentlərini modellərinizə asanlıqla daxil etməyə imkan verir. **Hugging Face Hub**-a baxın, istədiyiniz modeli seçin və təqdim edilmiş kod nümunələrindən istifadə edərək başlayın. Əvvəllər yalnız dil modelləri təklif edilirdi, lakin indi görüntü modelləri və daha çox komponent də əlavə edilmişdir.

16-cı fəsildə təbii dil işlənməsinə daha ətraflı qayıdacağıq. İndi isə Keras-ın görüntü emalı təbəqələrinə nəzər salaq.

Şəkil Əvvəlcədən Emalı Təbəqələri

Keras-ın **preprocessing API**-si üç əsas şəkil əvvəlcədən emalı təbəqəsini əhatə edir:

1. **tf.keras.layers.Resizing**
   Bu təbəqə giriş şəkillərini istədiyiniz ölçüyə yenidən ölçüləndirir. Məsələn, **Resizing(height=100, width=200)** hər bir şəkli 100 × 200 ölçüsünə dəyişir. Bu zaman şəkil təhrif oluna bilər. Əgər **crop_to_aspect_ratio=True** təyin etsəniz, təhrifin qarşısını almaq üçün şəkil hədəf ölçü nisbətinə uyğun olaraq kəsilir.
2. **tf.keras.layers.Rescaling**
   Piksel dəyərlərini yenidən ölçüləndirir. Məsələn, **Rescaling(scale=2/255, offset=-1)** 0 → 255 intervalındakı dəyərləri –1 → 1 intervalına dəyişir.
3. **tf.keras.layers.CenterCrop**
   Şəkilin yalnız mərkəz hissəsini, istənilən hündürlük və eni saxlamaqla kəsir.

Məsələn, gəlin bir neçə nümunə şəkli yükləyib, onları mərkəzdən kəsək. Bunun üçün Scikit-Learn-in **load_sample_images()** funksiyasından istifadə edəcəyik. Bu funksiya iki rəngli şəkil yükləyir: biri Çin məbədi, digəri isə bir çiçəkdir (bunun üçün Pillow kitabxanası tələb olunur. Əgər Colab istifadə edirsinizsə və ya quraşdırma təlimatlarına əməl etmisinizsə, Pillow artıq quraşdırılmışdır):

```
from sklearn.datasets import load_sample_images

images = load_sample_images()["images"]

crop_image_layer = tf.keras.layers.CenterCrop(height=100, width=100)

cropped_images = crop_image_layer(images)
```

Məlumatları Çoxaltma (Data Augmentation)

Keras həmçinin bir neçə məlumat çoxaltma təbəqəsi də təqdim edir, məsələn:

- **RandomCrop**
- **RandomFlip**
- **RandomTranslation**
- **RandomRotation**
- **RandomZoom**
- **RandomHeight**
- **RandomWidth**
- **RandomContrast**

Bu təbəqələr yalnız təlim zamanı aktivdir və giriş şəkillərinə təsadüfi çevrilmələr tətbiq edirlər (adları öz funksiyalarını izah edir). Məlumat çoxaltma üsulları, transformasiya olunmuş şəkillər real (çoxaltılmamış) şəkillərə bənzədiyi müddətcə, təlim dəstinin ölçüsünü süni şəkildə artıraraq performansın yaxşılaşmasına səbəb ola bilər. Şəkil emalını növbəti fəsildə daha ətraflı araşdıracağıq.

QEYD

Keras-ın əvvəlcədən emal təbəqələri TensorFlow-un aşağı səviyyəli API-lərinə əsaslanır. Məsələn:

- **Normalization** təbəqəsi orta və dispersiyanı hesablamaq üçün **tf.nn.moments()** istifadə edir.
- **Discretization** təbəqəsi **tf.raw_ops.Bucketize()**-dan istifadə edir.
- **CategoricalEncoding** təbəqəsi **tf.math.bincount()** istifadə edir.
- **IntegerLookup** və **StringLookup tf.lookup** paketinə əsaslanır.
- **Hashing** və **TextVectorization tf.strings** paketindəki bir neçə əməliyyatdan istifadə edir.
- **Embedding** təbəqəsi **tf.nn.embedding_lookup()** istifadə edir.
- Şəkil əvvəlcədən emalı təbəqələri isə **tf.image** paketinin əməliyyatlarından istifadə edir.

Əgər Keras-ın əvvəlcədən emal API-si ehtiyaclarınıza kifayət etmirsə, bəzən TensorFlow-un aşağı səviyyəli API-lərindən birbaşa istifadə etməli ola bilərsiniz.

İndi isə TensorFlow-da məlumatları asan və effektiv şəkildə yükləməyin başqa bir üsuluna baxaq.

TensorFlow Məlumat Dəstləri Layihəsi

TensorFlow Datasets (TFDS) layihəsi məşhur məlumat dəstlərini yükləməyi çox asanlaşdırır. Bu məlumat dəstləri kiçik ölçülü MNIST və Fashion MNIST-dən tutmuş, böyük həcmli **ImageNet** kimi dəstlərə qədər geniş bir spektri əhatə edir (bunun üçün kifayət qədər disk sahəsi lazım olacaq!). Siyahıya şəkil, mətn (tərcümə məlumat dəstləri daxil olmaqla), audio və video məlumat dəstləri, vaxt sıraları və daha çox növlər daxildir. Tam siyahını və hər bir məlumat dəstinin təsvirini **https://homl.info/tfds** saytından əldə edə bilərsiniz. Bundan əlavə, TFDS tərəfindən təqdim olunan məlumat dəstlərini araşdırmaq və daha yaxşı başa düşmək üçün **Know Your Data** alətindən istifadə edə bilərsiniz.

TFDS TensorFlow ilə birgə təqdim edilmir, lakin əgər Colab-da işləyirsinizsə və ya **https://homl.info/install**-dəki quraşdırma təlimatlarını izləmisinizsə, artıq quraşdırılıb. Bununla siz **tensorflow_datasets** kitabxanasını adətən **tfds** adı ilə idxal edə və sonra **tfds.load()** funksiyasını çağıraraq istədiyiniz məlumatı yükləyə bilərsiniz (əgər əvvəllər yüklənməyibsə). Bu funksiya məlumatları təlim və test üçün adətən iki dəst şəklində qaytarır (bu, seçdiyiniz məlumat dəstindən asılıdır). Məsələn, gəlin MNIST məlumat dəstini yükləyək:

```
import tensorflow_datasets as tfds

datasets = tfds.load(name="mnist")

mnist_train, mnist_test = datasets["train"], datasets["test"]
```

Bundan sonra, istədiyiniz hər hansı transformasiyanı (adətən qarışdırma, batch-ə bölmə və əvvəlcədən yükləmə) tətbiq edə bilərsiniz və modelinizi təlim etməyə hazırsınız. Sadə bir nümunə:

```
for batch in mnist_train.shuffle(10_000, seed=42).batch(32).prefetch(1):
    images = batch["image"]
    labels = batch["label"]
    # [...] şəkillərlə və etiketlərlə nəsə etmək
```

MƏSLƏHƏT:

load() funksiyası yüklənən faylları qarışdıra bilər: bunun üçün **shuffle_files=True** təyin edin. Ancaq bu, kifayət etməyə bilər, ona görə də təlim məlumatlarını əlavə qarışdırmaq daha yaxşıdır.

Datasetlərin Formatı

Hər bir məlumat dəstindəki element, xüsusiyyətləri və etiketləri ehtiva edən bir lüğətdir. Lakin **Keras**, hər bir elementi iki elementdən (xüsusiyyətlər və etiketlər) ibarət olan tuple kimi gözləyir. Siz dataset-i **map()** metodu ilə aşağıdakı kimi transformasiya edə bilərsiniz:

```
mnist_train = mnist_train.shuffle(buffer_size=10_000, seed=42).batch(32)
mnist_train = mnist_train.map(lambda items: (items["image"], items["label"]))
mnist_train = mnist_train.prefetch(1)
```

Ancaq bu prosesi sadələşdirmək üçün **load()** funksiyasını **as_supervised=True** parametri ilə çağıraraq məlumatları birbaşa tuple kimi qaytarmasını təmin edə bilərsiniz (bu, yalnız etiketlənmiş məlumat dəstləri üçün işləyir).

Məlumatın Bölünməsi

TFDS məlumatı bölmək üçün rahat **split** arqumentini təqdim edir. Məsələn, əgər təlim dəstinin ilk 90%-ni təlim üçün, qalan 10%-ni təsdiqləmə üçün və bütün test dəstini sınaq üçün istifadə etmək istəyirsinizsə, **split=["train[:90%]", "train[90%:]", "test"]** təyin edə bilərsiniz. **load()** funksiyası hər üç dəsti qaytaracaq. Tam bir nümunə:

```
train_set, valid_set, test_set = tfds.load(
    name="mnist",
    split=["train[:90%]", "train[90%:]", "test"],
```

```
        as_supervised=True
)
train_set = train_set.shuffle(buffer_size=10_000, seed=42).batch(32).prefetch(1)
valid_set = valid_set.batch(32).cache()
test_set = test_set.batch(32).cache()
tf.random.set_seed(42)
model = tf.keras.Sequential([
        tf.keras.layers.Flatten(input_shape=(28, 28)),
        tf.keras.layers.Dense(10, activation="softmax")
])
model.compile(
        loss="sparse_categorical_crossentropy",
        optimizer="nadam",
        metrics=["accuracy"]
)
history = model.fit(train_set, validation_data=valid_set, epochs=5)
test_loss, test_accuracy = model.evaluate(test_set)
```

Təbrik edirik! Bu texniki fəsli sona çatdırdınız. Neural şəbəkələrin abstrakt gözəlliyindən uzaq kimi görünsə də, dərin öyrənmə çox vaxt böyük miqdarda məlumatla işləməyi tələb edir və məlumatların yüklənməsi, təhlili və əvvəlcədən emalı sahəsində səmərəli olmaq vacib bacarıqdır. Növbəti fəsildə konvolyusiya neyron şəbəkələrinə baxacağıq. Bu şəbəkələr şəkil emalı və bir çox digər tətbiqlər üçün ən uğurlu neyron şəbəkə arxitekturalarındandır.

Tapşırıqlar

1. **Niyə tf.data API-dən istifadə etmək istəyərdiniz?**
2. **Böyük bir məlumat dəstini bir neçə fayla bölməyin üstünlükləri nələrdir?**
3. **Təlim zamanı giriş boru xəttinizin darboğaz olduğunu necə anlayarsınız? Bunun üçün nə edə bilərsiniz?**
4. **TFRecord faylına hər hansı ikili məlumatı yaza bilərsinizmi, yoxsa yalnız seriyalaşdırılmış protokol buferləri yazıla bilər?**
5. **Bütün məlumatlarınızı Example protobuf formatına çevirməyə niyə əziyyət çəkəsiniz? Öz protobuf tərifinizdən niyə istifadə etməyəsiniz?**

6. **TFRecords-dan istifadə edərkən kompressiyanı nə vaxt aktivləşdirmək istərdiniz? Niyə bunu sistematik şəkildə etməmək daha yaxşıdır?**
7. **Məlumatları fayllar yazılarkən, tf.data boru xəttində və ya model daxilində əvvəlcədən emal qatlarında emal edə bilərsiniz. Hər bir seçim üçün bir neçə müsbət və mənfi cəhəti sadalaya bilərsinizmi?**
8. **Kateqorial tam ədədi xüsusiyyətləri kodlaşdırmaq üçün bir neçə ümumi üsul adı çəkin. Bəs mətn üçün nə edərsiniz?**

---

9. **Fashion MNIST məlumat dəstini yükləyin** (10-cu fəsildə təqdim edilmişdir); onu təlim dəstinə, təsdiqləmə dəstinə və test dəstinə bölün; təlim dəstini qarışdırın və hər bir dəsti bir neçə TFRecord faylına saxlayın. Hər bir rekord iki xüsusiyyətə malik seriyalaşdırılmış Example protobuf olmalıdır:
    - Seriyalaşdırılmış şəkil (hər bir şəkli seriyalaşdırmaq üçün `tf.io.serialize_tensor()` istifadə edin)
    - Etiket. Sonra hər dəst üçün səmərəli bir dataset yaratmaq üçün `tf.data` istifadə edin. Sonda, hər giriş xüsusiyyətini standartlaşdırmaq üçün bir əvvəlcədən emal qatını daxil etməklə bu datasetləri təlim etmək üçün Keras modelindən istifadə edin. Giriş boru xəttini mümkün qədər səmərəli etmək üçün çalışın və TensorBoard istifadə edərək profilləmə məlumatlarını vizuallaşdırın.

---

10. **Bu tapşırıqda siz bir məlumat dəstini yükləyəcəksiniz, onu böləcəksiniz, onu səmərəli yükləmək və əvvəlcədən emal etmək üçün bir `tf.data.Dataset` yaradacaqsınız, sonra isə bir binary təsnifat modeli qurub təlim edəcəksiniz.**
    a. **Large Movie Review Dataset (IMDb-dən 50,000 film rəyi ehtiva edir) məlumat dəstini yükləyin.** Bu məlumatlar iki qovluqda təşkil olunub: `train` və `test`. Hər bir qovluqda müvafiq olaraq 12,500 müsbət və mənfi rəylər olan `pos` və `neg` alt qovluqları var. Hər bir rəy ayrıca mətn faylında saxlanılır. (Digər fayl və qovluqları, o cümlədən əvvəlcədən emal edilmiş bag-of-words versiyalarını, bu tapşırıqda nəzərə almayacağıq).
    b. **Test dəstini təsdiqləmə dəstinə (15,000) və test dəstinə (10,000) bölün.**
    c. **Hər dəst üçün səmərəli bir dataset yaratmaq üçün `tf.data` istifadə edin.**
    d. **Hər rəyi əvvəlcədən emal etmək üçün bir TextVectorization qatı istifadə edərək binary təsnifat modeli yaradın.**
    e. **Bir Embedding qatı əlavə edin və hər rəy üçün sözlərin sayının kvadrat kökünə vurulmuş orta embedding hesablayın** (16-cı fəsildə izah edilib). Bu yenidən ölçülmüş orta embedding daha sonra modelinizin qalan hissəsinə ötürülə bilər.
    f. **Modeli təlim edin və hansı dəqiqliyi əldə etdiyinizi görün. Təlimi mümkün qədər sürətli etmək üçün boru xətlərini optimallaşdırmağa çalışın.**
    g. **Eyni məlumat dəstini daha asan yükləmək üçün TFDS istifadə edin: `tfds.load("imdb_reviews")`.**

Bu tapşırıqların həlləri bu fəsilin notebookunun sonunda, https://homl.info/colab3 ünvanında mövcuddur.

---

1. Təsəvvür edin ki, solunuzda sıralanmış bir kart dəsti var: yuxarıdakı üç kartı götürüb qarışdırdığınızı, sonra isə təsadüfi birini seçib sağınıza qoyduğunuzu düşünün, qalan ikisini əlinizdə saxlayın. Sol tərəfdən başqa bir kart götürün, əlinizdəki üç kartı qarışdırıb yenidən təsadüfi birini seçin və sağınıza qoyun. Bu şəkildə bütün kartlardan keçib sağınızda yeni bir kart dəsti düzəltdiyinizdə, kartların tamamilə qarışdığını düşünürsünüzmü?
2. Ümumiyyətlə, yalnız bir dəsti əvvəlcədən yükləmək kifayətdir, amma bəzi hallarda bir neçə dəsti əvvəlcədən yükləməyə ehtiyac ola bilər. Alternativ olaraq, əvvəlcədən yükləmə üçün `tf.data.AUTOTUNE` istifadə edərək TensorFlow-un avtomatik qərar verməsinə icazə verə bilərsiniz.
3. Eksperimental olaraq `tf.data.experimental.prefetch_to_device()` funksiyasını nəzərdən keçirin, bu funksiya məlumatları birbaşa GPU-ya əvvəlcədən yükləyə bilər. Adında `experimental` olan hər hansı TensorFlow funksiyası və ya sinfi gələcək versiyalarda xəbərdarlıq etmədən dəyişdirilə bilər. Əgər eksperimental funksiya uğursuz olarsa, `experimental` sözünü çıxarmağı sınayın: bu funksiya əsas API-ya keçmiş ola bilər. Əks halda, zəhmət olmasa müvafiq notebooku yoxlayın, çünki mən onun aktual kodu ehtiva etməsini təmin edəcəyəm.
4. Protobuf obyektləri seriyalaşdırılmaq və ötürülmək üçün nəzərdə tutulduğundan, onlara mesajlar deyilir.
5. Bu fəsil, TFRecords istifadə etmək üçün protobuflar haqqında bilməli olduğunuz minimum əsas bilgiləri ehtiva edir. Protobuflar haqqında daha çox məlumat əldə etmək üçün https://homl.info/protobuf ünvanına baş çəkin.
6. Tomáš Mikolov və başqaları, “Sözlərin və İfadələrin Paylanmış Təmsil Edilməsi və Onların Kompozisiyası”, 26-cı Beynəlxalq Neyron Məlumatların Emalı Sistemləri Konfransı 2 (2013): 3111–3119.
7. Malvina Nissim və başqaları, “Ədalətli Daha Yaxşıdır: Həkim Kişi olduğu Kimi, Həkim də Qadındır”, arXiv preprint arXiv:1905.09866 (2019).
8. TensorFlow Hub TensorFlow ilə birgə gəlmir, amma əgər Colab-dan istifadə edirsinizsə və ya https://homl.info/install ünvanında quraşdırma təlimatlarına əməl etmisinizsə, artıq quraşdırılmış olacaq.
9. Dəqiq desək, cümlənin embedding-i söz embedding-lərinin ortalamasına bərabərdir və bu, cümlədəki sözlərin sayının kvadrat kökünə vurulur. Bu, təsadüfi vektorların ortalamasının sözlərin sayı artdıqca qısaldığını kompensasiya edir.
10. Böyük şəkillər üçün `tf.io.encode_jpeg()` istifadə edə bilərsiniz. Bu, çox yer qənaət edərdi, amma şəkil keyfiyyətinin bir qədər itirilməsi mümkündür.