CHAPTER 13

Loading and Preprocessing Data with TensorFlow

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as he or she writes—so you can take advantage of these technologies long before the official release of these titles. The following will be Chapter 13 in the final release of the book.

So far we have used only datasets that fit in memory, but Deep Learning systems are often trained on very large datasets that will not fit in RAM. Ingesting a large dataset and preprocessing it efficiently can be tricky to implement with other Deep Learning libraries, but TensorFlow makes it easy thanks to the *Data API*: you just create a dataset object, tell it where to get the data, then transform it in any way you want, and TensorFlow takes care of all the implementation details, such as multithreading, queuing, batching, prefetching, and so on.

Off the shelf, the Data API can read from text files (such as CSV files), binary files with fixed-size records, and binary files that use TensorFlow's TFRecord format, which supports records of varying sizes. TFRecord is a flexible and efficient binary format based on Protocol Buffers (an open source binary format). The Data API also has support for reading from SQL databases. Moreover, many Open Source extensions are available to read from all sorts of data sources, such as Google's BigQuery service.

However, reading huge datasets efficiently is not the only difficulty: the data also needs to be preprocessed. Indeed, it is not always composed strictly of convenient numerical fields: sometimes there will be text features, categorical features, and so on. To handle this, TensorFlow provides the *Features API*: it lets you easily convert these features to numerical features that can be consumed by your neural network. For

example, categorical features with a large number of categories (such as cities, or words) can be encoded using *embeddings* (as we will see, an embedding is a trainable dense vector that represents a category).

Both the Data API and the Features API work seamlessly with tf.keras.

In this chapter, we will cover the Data API, the TFRecord format and the Features API in detail. We will also take a quick look at a few related projects from TensorFlow's ecosystem:

- TF Transform (*tf.Transform*) makes it possible to write a single preprocessing function that can be run both in batch mode on your full training set, before training (to speed it up), and then exported to a TF Function and incorporated into your trained model, so that once it is deployed in production, it can take care of preprocessing new instances on the fly.
- TF Datasets (TFDS) provides a convenient function to download many common datasets of all kinds, including large ones like ImageNet, and it provides convenient dataset objects to manipulate them using the Data API.

So let's get started!

The Data API

The whole Data API revolves around the concept of a *dataset*: as you might suspect, this represents a sequence of data items. Usually you will use datasets that gradually read data from disk, but for simplicity let's just create a dataset entirely in RAM using `tf.data.Dataset.from_tensor_slices()`:

```
>>> X = tf.range(10)  # any data tensor
>>> dataset = tf.data.Dataset.from_tensor_slices(X)
>>> dataset
<TensorSliceDataset shapes: (), types: tf.int32>
```

The `from_tensor_slices()` function takes a tensor and creates a `tf.data.Dataset` whose elements are all the slices of `X` (along the first dimension), so this dataset contains 10 items: tensors 0, 1, 2, …, 9. In this case we would have obtained the same dataset if we had used `tf.data.Dataset.range(10)`.

You can simply iterate over a dataset's items like this:

```
>>> 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(9, shape=(), dtype=int32)
```

Chaining Transformations

Once you have a dataset, you can apply all sorts of transformations to it by calling its transformation methods. Each method returns a new dataset, so you can chain transformations like this (this chain is illustrated in Figure 13-1):

```
>>> 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)
```

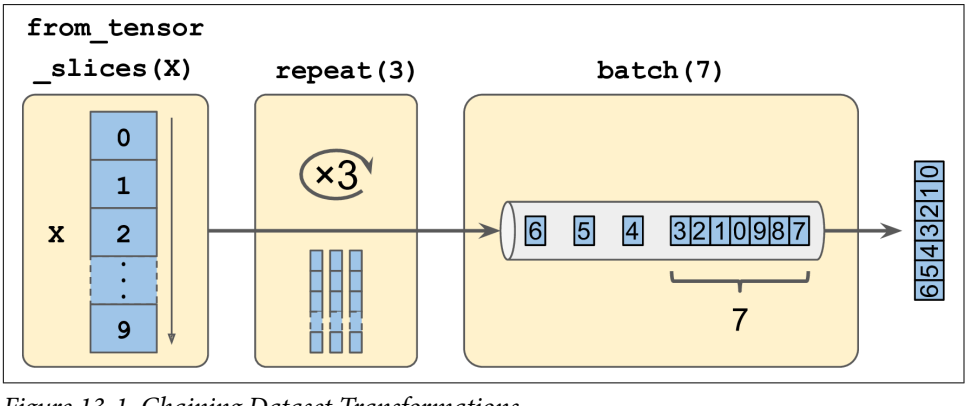

Figure 13-1. Chaining Dataset Transformations

In this example, we first call the `repeat()` method on the original dataset, and it returns a new dataset that will repeat the items of the original dataset 3 times. Of course, this will not copy the whole data in memory 3 times! In fact, if you call this method with no arguments, the new dataset will repeat the source dataset forever. Then we call the `batch()` method on this new dataset, and again this creates a new dataset. This one will group the items of the previous dataset in batches of 7 items. Finally, we iterate over the items of this final dataset. As you can see, the `batch()` method had to output a final batch of size 2 instead of 7, but you can call it with `drop_remainder=True` if you want it to drop this final batch so that all batches have the exact same size.

The dataset methods do *not* modify datasets, they create new ones, so make sure to keep a reference to these new datasets (e.g., `dataset = ...`), or else nothing will happen.

You can also apply any transformation you want to the items by calling the `map()` method. For example, this creates a new dataset with all items doubled:

```
>>> dataset = dataset.map(lambda x: x * 2) # Items: [0,2,4,6,8,10,12]
```

This function is the one you will call to apply any preprocessing you want to your data. Sometimes, this will include computations that can be quite intensive, such as reshaping or rotating an image, so you will usually want to spawn multiple threads to speed things up: it's as simple as setting the `num_parallel_calls` argument.

While the `map()` applies a transformation to each item, the `apply()` method applies a transformation to the dataset as a whole. For example, the following code "unbatches" the dataset, by applying the `unbatch()` function to the dataset (this function is currently experimental, but it will most likely move to the core API in a future release). Each item in the new dataset will be a single integer tensor instead of a batch of 7 integers:

```
>>> dataset = dataset.apply(tf.data.experimental.unbatch()) # Items: 0,2,4,...
```

It is also possible to simply filter the dataset using the `filter()` method:

```
>>> dataset = dataset.filter(lambda x: x < 10) # Items: 0 2 4 6 8 0 2 4 6...
```

You will often want to look at just a few items from a dataset. You can use the `take()` method for that:

```
>>> for item in dataset.take(3):
...     print(item)
...
tf.Tensor(0, shape=(), dtype=int64)
tf.Tensor(2, shape=(), dtype=int64)
tf.Tensor(4, shape=(), dtype=int64)
```

Shuffling the Data

As you know, Gradient Descent works best when the instances in the training set are independent and identically distributed (see Chapter 4). A simple way to ensure this is to shuffle the instances. For this, you can just use the `shuffle()` method. It will create a new dataset that will start by filling up a buffer with the first items of the source dataset, then whenever it is asked for an item, it will pull one out randomly from the buffer, and replace it with a fresh one from the source dataset, until it has iterated entirely through the source dataset. At this point it continues to pull out items randomly from the buffer until it is empty. You must specify the buffer size, and

it is important to make it large enough or else shuffling will not be very efficient.[1] However, obviously do not exceed the amount of RAM you have, and even if you have plenty of it, there's no need to go well beyond the dataset's size. You can provide a random seed if you want the same random order every time you run your program.

```
>>> dataset = tf.data.Dataset.range(10).repeat(3) # 0 to 9, three times
>>> 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)
```

If you call `repeat()` on a shuffled dataset, by default it will generate a new order at every iteration. This is generally a good idea, but if you prefer to reuse the same order at each iteration (e.g., for tests or debugging), you can set `reshuffle_each_iteration=False`.

For a large dataset that does not fit in memory, this simple shuffling-buffer approach may not be sufficient, since the buffer will be small compared to the dataset. One solution is to shuffle the source data itself (for example, on Linux you can shuffle text files using the `shuf` command). This will definitely improve shuffling a lot! However, even if the source data is shuffled, you will usually want to shuffle it some more, or else the same order will be repeated at each epoch, and the model may end up being biased (e.g., due to some spurious patterns present by chance in the source data's order). To shuffle the instances some more, a common approach is to split the source data into multiple files, then read them in a random order during training. However, instances located in the same file will still end up close to each other. To avoid this you can pick multiple files randomly, and read them simultaneously, interleaving their lines. Then on top of that you can add a shuffling buffer using the `shuffle()` method. If all this sounds like a lot of work, don't worry: the Data API actually makes all this possible in just a few lines of code. Let's see how to do this.

1 Imagine a sorted deck of cards on your left: suppose you just take the top 3 cards and shuffle them, then pick one randomly and put it to your right, keeping the other 2 in your hands. Take another card on your left, shuffle the 3 cards in your hands and pick one of them randomly, and put it on your right. When you are done going through all the cards like this, you will have a deck of cards on your right: do you think it will be perfectly shuffled?

Interleaving Lines From Multiple Files

First, let's suppose that you loaded the California housing dataset, you shuffled it (unless it was already shuffled), you split it into a training set, a validation set and a test set, then you split each set into many CSV files that each look like this (each row contains 8 input features plus the target median house value):

```
MedInc,HouseAge,AveRooms,AveBedrms,Popul,AveOccup,Lat,Long,MedianHouseValue
3.5214,15.0,3.0499,1.1065,1447.0,1.6059,37.63,-122.43,1.442
5.3275,5.0,6.4900,0.9910,3464.0,3.4433,33.69,-117.39,1.687
3.1,29.0,7.5423,1.5915,1328.0,2.2508,38.44,-122.98,1.621
[...]
```

Let's also suppose `train_filepaths` contains the list of file paths (and you also have `valid_filepaths` and `test_filepaths`):

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

Now let's create a dataset containing only these file paths:

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

By default, the `list_files()` function returns a dataset that shuffles the file paths. In general this is a good thing, but you can set `shuffle=False` if you do not want that, for some reason.

Next, we can call the `interleave()` method to read from 5 files at a time and interleave their lines (skipping the first line of each file, which is the header row, using the `skip()` method):

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

The `interleave()` method will create a dataset that will pull 5 file paths from the `filepath_dataset`, and for each one it will call the function we gave it (a lambda in this example) to create a new dataset, in this case a `TextLineDataset`. It will then cycle through these 5 datasets, reading one line at a time from each until all datasets are out of items. Then it will get the next 5 file paths from the `filepath_dataset`, and interleave them the same way, and so on until it runs out of file paths.

For interleaving to work best, it is preferable to have files of identical length, or else the end of the longest files will not be interleaved.

By default, `interleave()` does not use parallelism, it just reads one line at a time from each file, sequentially. However, if you want it to actually read files in parallel, you can set the `num_parallel_calls` argument to the number of threads you want. You can even set it to `tf.data.experimental.AUTOTUNE` to make TensorFlow choose the right number of threads dynamically based on the available CPU (however, this is an experimental feature for now). Let's look at what the dataset contains now:

```
>>> for line in dataset.take(5):
...     print(line.numpy())
...
b'4.2083,44.0,5.3232,0.9171,846.0,2.3370,37.47,-122.2,2.782'
b'4.1812,52.0,5.7013,0.9965,692.0,2.4027,33.73,-118.31,3.215'
b'3.6875,44.0,4.5244,0.9930,457.0,3.1958,34.04,-118.15,1.625'
b'3.3456,37.0,4.5140,0.9084,458.0,3.2253,36.67,-121.7,2.526'
b'3.5214,15.0,3.0499,1.1065,1447.0,1.6059,37.63,-122.43,1.442'
```

These are the first rows (ignoring the header row) of 5 CSV files, chosen randomly. Looks good! But as you can see, these are just byte strings, we need to parse them, and also scale the data.

Preprocessing the Data

Let's implement a small function that will perform this preprocessing:

```
X_mean, X_std = [...] # mean and scale of each feature in the training set
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
```

Let's walk through this code:

- First, we assume that you have precomputed the mean and standard deviation of each feature in the training set. `X_mean` and `X_std` are just 1D tensors (or NumPy arrays) containing 8 floats, one per input feature.
- The `preprocess()` function takes one CSV line, and starts by parsing it. For this, it uses the `tf.io.decode_csv()` function, which takes two arguments: the first is the line to parse, and the second is an array containing the default value for each column in the CSV file. This tells TensorFlow not only the default value for each column, but also the number of columns and the type of each column. In this example, we tell it that all feature columns are floats and missing values should default to 0, but we provide an empty array of type `tf.float32` as the default value for the last column (the target): this tells TensorFlow that this column con-

tains floats, but that there is no default value, so it will raise an exception if it encounters a missing value.

- The `decode_csv()` function returns a list of scalar tensors (one per column) but we need to return 1D tensor arrays. So we call `tf.stack()` on all tensors except for the last one (the target): this will stack these tensors into a 1D array. We then do the same for the target value (this makes it a 1D tensor array with a single value, rather than a scalar tensor).
- Finally, we scale the input features by subtracting the feature means and then dividing by the feature standard deviations, and we return a tuple containing the scaled features and the target.

Let's test this preprocessing function:

```
>>> preprocess(b'4.2083,44.0,5.3232,0.9171,846.0,2.3370,37.47,-122.2,2.782')
(<tf.Tensor: id=6227, 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: [...], numpy=array([2.782], dtype=float32)>)
```

We can now apply this preprocessing function to the dataset.

Putting Everything Together

To make the code reusable, let's put together everything we have discussed so far into a small helper function: it will create and return a dataset that will efficiently load California housing data from multiple CSV files, then shuffle it, preprocess it and batch it (see Figure 13-2):

```
def csv_reader_dataset(filepaths, repeat=None, 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)
```

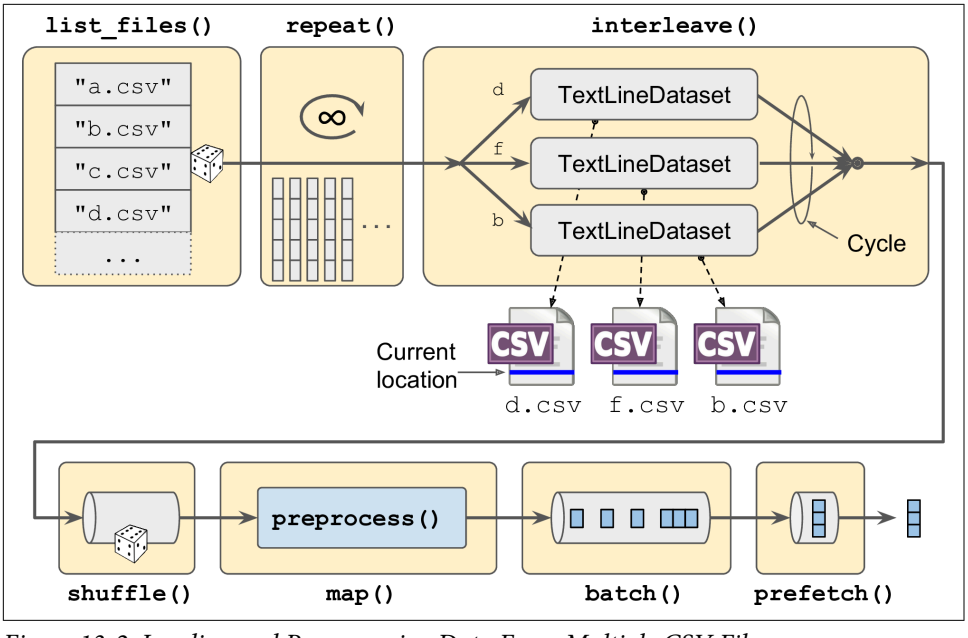

Figure 13-2. Loading and Preprocessing Data From Multiple CSV Files

Everything should make sense in this code, except the very last line (`prefetch(1)`), which is actually quite important for performance.

Prefetching

By calling `prefetch(1)` at the end, we are creating a dataset that will do its best to always be one batch ahead[2]. In other words, while our training algorithm is working on one batch, the dataset will already be working in parallel on getting the next batch ready. This can improve performance dramatically, as is illustrated on Figure 13-3. If we also ensure that loading and preprocessing are multithreaded (by setting `num_parallel_calls` when calling `interleave()` and `map()`), we can exploit multiple cores on the CPU and hopefully make preparing one batch of data shorter than running a training step on the GPU: this way the GPU will be almost 100% utilized (except for the data transfer time from the CPU to the GPU), and training will run much faster.

2 In general, just prefetching one batch is fine, but in some cases you may need to prefetch a few more. Alternatively, you can let TensorFlow decide automatically by passing `tf.data.experimental.AUTOTUNE` (this is an experimental feature for now).

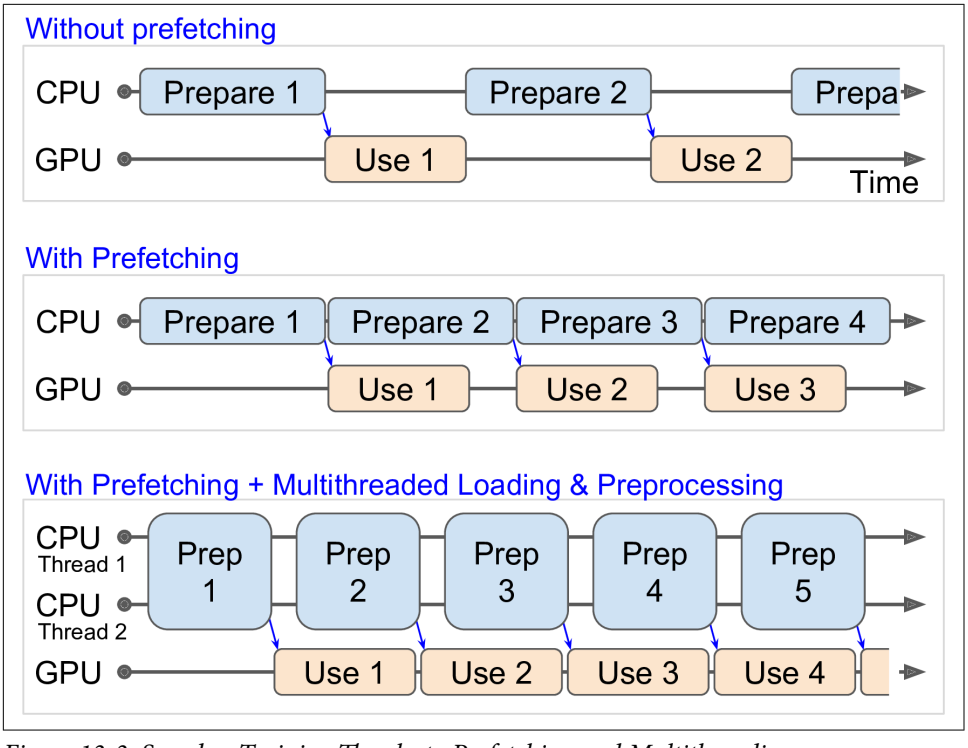

Figure 13-3. Speedup Training Thanks to Prefetching and Multithreading

If you plan to purchase a GPU card, its processing power and its memory size are of course very important (in particular, a large RAM is crucial for computer vision), but its *memory bandwidth* is just as important as the processing power to get good performance: this is the number of gigabytes of data it can get in or out of its RAM per second.

With that, you can now build efficient input pipelines to load and preprocess data from multiple text files. We have discussed the most common dataset methods, but there are a few more you may want to look at: `concatenate()`, `zip()`, `window()`, `reduce()`, `cache()`, `shard()`, `flat_map()` and `padded_batch()`. There are also a couple more class methods: `from_generator()` and `from_tensors()`, which create a new dataset from a Python generator or a list of tensors respectively. Please check the API documentation for more details. Also note that there are experimental features available in `tf.data.experimental`, many of which will most likely make it to the core API in future releases (e.g., check out the `CsvDataset` class and the `SqlDataset` classes).

Using the Dataset With tf.keras

Now we can use the `csv_reader_dataset()` function to create a dataset for the training set (ensuring it repeats the data forever), the validation set and the test set:

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

And now we can simply build and train a Keras model using these datasets.[3] All we need to do is to call the `fit()` method with the datasets instead of `X_train` and `y_train`, and specify the number of steps per epoch for each set:[4]

```
model = keras.models.Sequential([...])
model.compile([...])
model.fit(train_set, steps_per_epoch=len(X_train) // batch_size, epochs=10,
          validation_data=valid_set,
          validation_steps=len(X_valid) // batch_size)
```

Similarly, we can pass a dataset to the `evaluate()` and `predict()` methods (and again specify the number of steps per epoch):

```
model.evaluate(test_set, steps=len(X_test) // batch_size)
model.predict(new_set, steps=len(X_new) // batch_size)
```

Unlike the other sets, the `new_set` will usually not contain labels (if it does, Keras will just ignore them). Note that in all these cases, you can still use NumPy arrays instead of datasets if you want (but of course they need to have been loaded and preprocessed first).

If you want to build your own custom training loop (as in Chapter 12), you can just iterate over the training set, very naturally:

```
for X_batch, y_batch in train_set:
    [...] # perform one gradient descent step
```

In fact, it is even possible to create a tf.function (see Chapter 12) that performs the whole training loop![5]

```
@tf.function
def train(model, optimizer, loss_fn, n_epochs, [...]):
    train_set = csv_reader_dataset(train_filepaths, repeat=n_epochs, [...])
    for X_batch, y_batch in train_set:
        with tf.GradientTape() as tape:
```

3 Support for datasets is specific to tf.keras, it will not work on other implementations of the Keras API.

4 The number of steps per epoch is optional if the dataset just goes through the data once, but if you do not specify it, the progress bar will not be displayed during the first epoch.

5 Note that for now the dataset must be created within the TF Function. This may be fixed by the time you read these lines (see TensorFlow issue #25414).

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

Congratulations, you now know how to build powerful input pipelines using the Data API! However, so far we have used CSV files, which are common, simple and convenient, but they are not really efficient, and they do not support large or complex data structures very well, such as images or audio. So let's use TFRecords instead.

If you are happy with CSV files (or whatever other format you are using), you do not *have* to use TFRecords. As the saying goes, if it ain't broke, don't fix it! TFRecords are useful when the bottleneck during training is loading and parsing the data.

The TFRecord Format

The TFRecord format is TensorFlow's preferred format for storing large amounts of data and reading it efficiently. It is a very simple binary format that just contains a sequence of binary records of varying sizes (each record just has a length, a CRC checksum to check that the length was not corrupted, then the actual data, and finally a CRC checksum for the data). You can easily create a TFRecord file using the `tf.io.TFRecordWriter` class:

```
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")
```

And you can then use a `tf.data.TFRecordDataset` to read one or more TFRecord files:

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

This will output:

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

By default, a `TFRecordDataset` will read files one by one, but you can make it read multiple files in parallel and interleave their records by setting `num_parallel_reads`. Alternatively, you could obtain the same result by using `list_files()` and `interleave()` as we did earlier to read multiple CSV files.

Compressed TFRecord Files

It can sometimes be useful to compress your TFRecord files, especially if they need to be loaded via a network connection. You can create a compressed TFRecord file by setting the `options` argument:

```
options = tf.io.TFRecordOptions(compression_type="GZIP")
with tf.io.TFRecordWriter("my_compressed.tfrecord", options) as f:
  [...]
```

When reading a compressed TFRecord file, you need to specify the compression type:

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

A Brief Introduction to Protocol Buffers

Even though each record can use any binary format you want, TFRecord files usually contain serialized Protocol Buffers (also called *protobufs*). This is a portable, extensible and efficient binary format developed at Google back in 2001 and Open Sourced in 2008, and they are now widely used, in particular in gRPC, Google's remote procedure call system. Protocol Buffers are defined using a simple language that looks like this:

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

This definition says we are using the protobuf format version 3, and it specifies that each `Person` object[6] may (optionally) have a `name` of type `string`, an `id` of type `int32`, and zero or more `email` fields, each of type `string`. The numbers 1, 2 and 3 are the field identifiers: they will be used in each record's binary representation. Once you have a definition in a `.proto` file, you can compile it. This requires `protoc`, the protobuf compiler, to generate access classes in Python (or some other language). Note that the protobuf definitions we will use have already been compiled for you, and their Python classes are part of TensorFlow, so you will not need to use `protoc`. All you need to know is how to use protobuf access classes in Python. To illustrate the basics, let's look at a simple example that uses the access classes generated for the `Person` protobuf (the code is explained in the comments):

```
>>> from person_pb2 import Person  # import the generated access class
>>> person = Person(name="Al", id=123, email=["a@b.com"])  # create a Person
>>> print(person)  # display the Person
```

6 Since protobuf objects are meant to be serialized and transmitted, they are called *messages*.

```
name: "Al"
id: 123
email: "a@b.com"
>>> person.name  # read a field
"Al"
>>> person.name = "Alice"  # modify a field
>>> person.email[0]  # repeated fields can be accessed like arrays
"a@b.com"
>>> person.email.append("c@d.com")  # add an email address
>>> s = person.SerializeToString()  # serialize the object to a byte string
>>> s
b'\n\x05Alice\x10{\x1a\x07a@b.com\x1a\x07c@d.com'
>>> person2 = Person()  # create a new Person
>>> person2.ParseFromString(s)  # parse the byte string (27 bytes long)
27
>>> person == person2  # now they are equal
True
```

In short, we import the `Person` class generated by `protoc`, we create an instance and we play with it, visualizing it, reading and writing some fields, then we serialize it using the `SerializeToString()` method. This is the binary data that is ready to be saved or transmitted over the network. When reading or receiving this binary data, we can parse it using the `ParseFromString()` method, and we get a copy of the object that was serialized.[7]

We could save the serialized `Person` object to a TFRecord file, then we could load and parse it: everything would work fine. However, `SerializeToString()` and `ParseFromString()` are not TensorFlow operations (and neither are the other operations in this code), so they cannot be included in a TensorFlow Function (except by wrapping them in a `tf.py_function()` operation, which would make the code slower and less portable, as we saw in Chapter 12). Fortunately, TensorFlow does include special protobuf definitions for which it provides parsing operations.

TensorFlow Protobufs

The main protobuf typically used in a TFRecord file is the `Example` protobuf, which represents one instance in a dataset. It contains a list of named features, where each feature can either be a list of byte strings, a list of floats or a list of integers. Here is the protobuf definition:

```
syntax = "proto3";
message BytesList { repeated bytes value = 1; }
message FloatList { repeated float value = 1 [packed = true]; }
message Int64List { repeated int64 value = 1 [packed = true]; }
```

7 This chapter contains the bare minimum you need to know about protobufs to use TFRecords. To learn more about protobufs, please visit *https://homl.info/protobuf*.

```
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; };
```

The definitions of `BytesList`, `FloatList` and `Int64List` are straightforward enough (`[packed = true]` is used for repeated numerical fields, for a more efficient encoding). A `Feature` either contains a `BytesList`, a `FloatList` or an `Int64List`. A `Features` (with an s) contains a dictionary that maps a feature name to the corresponding feature value. And finally, an `Example` just contains a `Features` object.[8] Here is how you could create a `tf.train.Example` representing the same person as earlier, and write it to TFRecord file:

```
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"]))
        }))
```

The code is a bit verbose and repetitive, but it's rather straightforward (and you could easily wrap it inside a small helper function). Now that we have an `Example` protobuf, we can serialize it by calling its `SerializeToString()` method, then write the resulting data to a TFRecord file:

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

Normally you would write much more than just one example! Typically, you would create a conversion script that reads from your current format (say, CSV files), creates an `Example` protobuf for each instance, serializes them and saves them to several TFRecord files, ideally shuffling them in the process. This requires a bit of work, so once again make sure it is really necessary (perhaps your pipeline works fine with CSV files).

8 Why was `Example` even defined since it contains no more than a `Features` object? Well, TensorFlow may one day decide to add more fields to it. As long as the new `Example` definition still contains the `features` field, with the same id, it will be backward compatible. This extensibility is one of the great features of protobufs.

Now that we have a nice TFRecord file containing a serialized `Example`, let's try to load it.

Loading and Parsing Examples

To load the serialized `Example` protobufs, we will use a `tf.data.TFRecordDataset` once again, and we will parse each `Example` using `tf.io.parse_single_example()`. This is a TensorFlow operation so it can be included in a TF Function. It requires at least two arguments: a string scalar tensor containing the serialized data, and a description of each feature. The description is a dictionary that maps each feature name to either a `tf.io.FixedLenFeature` descriptor indicating the feature's shape, type and default value, or a `tf.io.VarLenFeature` descriptor indicating only the type (if the length may vary, such as for the `"emails"` feature). For example:

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

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

The fixed length features are parsed as regular tensors, but the variable length features are parsed as sparse tensors. You can convert a sparse tensor to a dense tensor using `tf.sparse.to_dense()`, but in this case it is simpler to just access its values:

```
>>> 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'], [...])>
```

A `BytesList` can contain any binary data you want, including any serialized object. For example, you can use `tf.io.encode_jpeg()` to encode an image using the JPEG format, and put this binary data in a `BytesList`. Later, when your code reads the TFRecord, it will start by parsing the `Example`, then you will need to call `tf.io.decode_jpeg()` to parse the data and get the original image (or you can use `tf.io.decode_image()`, which can decode any BMP, GIF, JPEG or PNG image). You can also store any tensor you want in a `BytesList` by serializing the tensor using `tf.io.serialize_tensor()`, then putting the resulting byte string in a `BytesList` feature. Later, when you parse the TFRecord, you can parse this data using `tf.io.parse_tensor()`.

Instead of parsing examples one by one using `tf.io.parse_single_example()`, you may want to parse them batch by batch using `tf.io.parse_example()`:

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

As you can see, the `Example` proto will probably be sufficient for most use cases. However, it may be a bit cumbersome to use when you are dealing with lists of lists. For example, suppose you want to classify text documents. Each document may be represented as a list of sentences, where each sentence is represented as a list of words. And perhaps each document also has a list of comments, where each comment is also represented as a list of words. Moreover, there may be some contextual data as well, such as the document's author, title and publication date. TensorFlow's `SequenceExample` protobuf is designed for such use cases.

Handling Lists of Lists Using the `SequenceExample` Protobuf

Here is the definition of the `SequenceExample` protobuf:

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

A `SequenceExample` contains a `Features` object for the contextual data and a `FeatureLists` object which contains one or more named `FeatureList` objects (e.g., a `FeatureList` named `"content"` and another named `"comments"`). Each `FeatureList` just contains a list of `Feature` objects, each of which may be a list of byte strings, a list of 64-bit integers or a list of floats (in this example, each `Feature` would represent a sentence or a comment, perhaps in the form of a list of word identifiers). Building a `SequenceExample`, serializing it and parsing it is very similar to building, serializing and parsing an `Example`, but you must use `tf.io.parse_single_sequence_example()` to parse a single `SequenceExample` or `tf.io.parse_sequence_example()` to parse a batch, and both functions return a tuple containing the context features (as a dictionary) and the feature lists (also as a dictionary). If the feature lists contain sequences of varying sizes (as in the example above), you may want to convert them to ragged tensors using `tf.RaggedTensor.from_sparse()` (see the notebook for the full code):

```
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"])
```

Now that you know how to efficiently store, load and parse data, the next step is to prepare it so that it can be fed to a neural network. This means converting all features

into numerical features (ideally not too sparse), scaling them, and more. In particular, if your data contains categorical features or text features, they need to be converted to numbers. For this, the *Features API* can help.

The Features API

Preprocessing your data can be performed in many ways: it can be done ahead of time when preparing your data files, using any tool you like. Or you can preprocess your data on the fly when loading it with the Data API (e.g., using the dataset's `map()` method, as we saw earlier). Or you can include a preprocessing layer directly in your model. Whichever solution you prefer, the Features API can help you: it is a set of functions available in the `tf.feature_column` package, which let you define how each feature (or group of features) in your data should be preprocessed (therefore you can think of this API as the analog of Scikit-Learn's `ColumnTransformer` class). We will start by looking at the different types of columns available, and then we will look at how to use them.

Let's go back to the variant of the California housing dataset that we used in Chapter 2, since it includes a categorical feature and missing data. Here is a simple numerical column named `"housing_median_age"`:

```
housing_median_age = tf.feature_column.numeric_column("housing_median_age")
```

Numeric columns let you specify a normalization function using the `normalizer_fn` argument. For example, let's tweak the `"housing_median_age"` column to define how it should be scaled. Note that this requires computing ahead of time the mean and standard deviation of this feature in the training set:

```
age_mean, age_std = X_mean[1], X_std[1]  # The median age is column in 1
housing_median_age = tf.feature_column.numeric_column(
    "housing_median_age", normalizer_fn=lambda x: (x - age_mean) / age_std)
```

In some cases, it might improve performance to bucketize some numerical features, effectively transforming a numerical feature into a categorical feature. For example, let's create a bucketized column based on the `median_income` column, with 5 buckets: less than 1.5 ($15,000), then 1.5 to 3, 3 to 4.5, 4.5 to 6., and above 6. (notice that when you specify 4 boundaries, there are actually 5 buckets):

```
median_income = tf.feature_column.numeric_column("median_income")
bucketized_income = tf.feature_column.bucketized_column(
    median_income, boundaries=[1.5, 3., 4.5, 6.])
```

If the `median_income` feature is equal to, say, 3.2, then the `bucketized_income` feature will automatically be equal to 2 (i.e., the index of the corresponding income bucket). Choosing the right boundaries can be somewhat of an art, but one approach is to just use percentiles of the data (e.g., the 10th percentile, the 20th percentile, and so on). If a feature is *multimodal*, meaning it has separate peaks in its distribution, you may

want to define a bucket for each mode, placing the boundaries in between the peaks. Whether you use the percentiles or the modes, you need to analyze the distribution of your data ahead of time, just like we had to measure the mean and standard deviation ahead of time to normalize the `housing_median_age` column.

Categorical Features

For categorical features such as `ocean_proximity`, there are several options. If it is already represented as a category ID (i.e., an integer from 0 to the max ID), then you can use the `categorical_column_with_identity()` function (specifying the max ID). If not, and you know the list of all possible categories, then you can use `categorical_column_with_vocabulary_list()`:

```
ocean_prox_vocab = ['<1H OCEAN', 'INLAND', 'ISLAND', 'NEAR BAY', 'NEAR OCEAN']
ocean_proximity = tf.feature_column.categorical_column_with_vocabulary_list(
    "ocean_proximity", ocean_prox_vocab)
```

If you prefer to have TensorFlow load the vocabulary from a file, you can call `categorical_column_with_vocabulary_file()` instead. As you might expect, these two functions will simply map each category to its index in the vocabulary (e.g., *NEAR BAY* will be mapped to 3), and unknown categories will be mapped to -1.

For categorical columns with a large vocabulary (e.g., for zipcodes, cities, words, products, users, etc.), it may not be convenient to get the full list of possible categories, or perhaps categories may be added or removed so frequently that using category indices would be too unreliable. In this case, you may prefer to use a `categorical_column_with_hash_bucket()`. If we had a `"city"` feature in the dataset, we could encode it like this:

```
city_hash = tf.feature_column.categorical_column_with_hash_bucket(
    "city", hash_bucket_size=1000)
```

This feature will compute a hash for each category (i.e., for each city), modulo the number of hash buckets (`hash_bucket_size`). You must set the number of buckets high enough to avoid getting too many collisions (i.e., different categories ending up in the same bucket), but the higher you set it, the more RAM will be used (by the embedding table, as we will see shortly).

Crossed Categorical Features

If you suspect that two (or more) categorical features are more meaningful when used jointly, then you can create a *crossed column*. For example, suppose people are particularly fond of old houses inland and new houses near the ocean, then it might help to

create a bucketized column for the `housing_median_age` feature[9], and cross it with the `ocean_proximity` column. The crossed column will compute a hash of every age & ocean proximity combination it comes across, modulo the `hash_bucket_size`, and this will give it the cross category ID. You may then choose to use only this crossed column in your model, or also include the individual columns.

```
bucketized_age = tf.feature_column.bucketized_column(
    housing_median_age, boundaries=[-1., -0.5, 0., 0.5, 1.]) # age was scaled
age_and_ocean_proximity = tf.feature_column.crossed_column(
    [bucketized_age, ocean_proximity], hash_bucket_size=100)
```

Another common use case for crossed columns is to cross latitude and longitude into a single categorical feature: you start by bucketizing the latitude and longitude, for example into 20 buckets each, then you cross these bucketized features into a `location` column. This will create a 20×20 grid over California, and each cell in the grid will correspond to one category:

```
latitude = tf.feature_column.numeric_column("latitude")
longitude = tf.feature_column.numeric_column("longitude")
bucketized_latitude = tf.feature_column.bucketized_column(
    latitude, boundaries=list(np.linspace(32., 42., 20 - 1)))
bucketized_longitude = tf.feature_column.bucketized_column(
    longitude, boundaries=list(np.linspace(-125., -114., 20 - 1)))
location = tf.feature_column.crossed_column(
    [bucketized_latitude, bucketized_longitude], hash_bucket_size=1000)
```

Encoding Categorical Features Using One-Hot Vectors

No matter which option you choose to build a categorical feature (categorical columns, bucketized columns or crossed columns), it must be encoded before you can feed it to a neural network. There are two options to encode a categorical feature: one-hot vectors or *embeddings*. For the first option, simply use the `indicator_column()` function:

```
ocean_proximity_one_hot = tf.feature_column.indicator_column(ocean_proximity)
```

A one-hot vector encoding has the size of the vocabulary length, which is fine if there are just a few possible categories, but if the vocabulary is large, you will end up with too many inputs fed to your neural network: it will have too many weights to learn and it will probably not perform very well. In particular, this will typically be the case when you use hash buckets. In this case, you should probably encode them using *embeddings* instead.

9 Since the `housing_median_age` feature was normalized, the boundaries are for normalized ages.

As a rule of thumb (but your mileage may vary!), if the number of categories is lower than 10, then one-hot encoding is generally the way to go. If the number of categories is greater than 50 (which is often the case when you use hash buckets), then embeddings are usually preferable. In between 10 and 50 categories, you may want to experiment with both options and see which one works best for your use case. Also, embeddings typically require more training data, unless you can reuse pretrained embeddings.

Encoding Categorical Features Using Embeddings

An embedding is a trainable dense vector that represents a category. By default, embeddings are initialized randomly, so for example the `"NEAR BAY"` category could be represented initially by a random vector such as `[0.131, 0.890]`, while the `"NEAR OCEAN"` category may be represented by another random vector such as `[0.631, 0.791]` (in this example, we are using 2D embeddings, but the number of dimensions is a hyperparameter you can tweak). Since these embeddings are trainable, they will gradually improve during training, and as they represent fairly similar categories, Gradient Descent will certainly end up pushing them closer together, while it will tend to move them away from the `"INLAND"` category's embedding (see Figure 13-4). Indeed, the better the representation, the easier it will be for the neural network to make accurate predictions, so training tends to make embeddings useful representations of the categories. This is called *representation learning* (we will see other types of representation learning in ???).

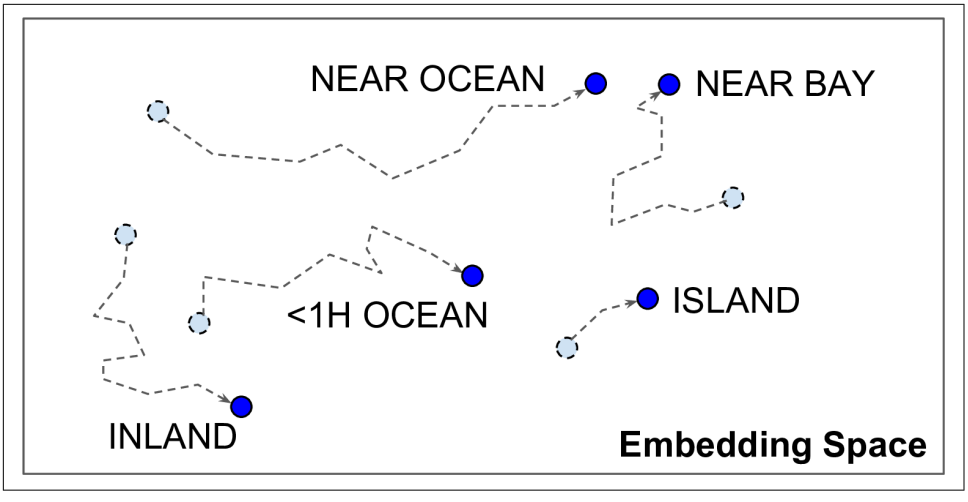

Figure 13-4. Embeddings Will Gradually Improve During Training

Word Embeddings

Not only will embeddings generally be useful representations for the task at hand, but quite often these same embeddings can be reused successfully for other tasks as well. The most common example of this is *word embeddings* (i.e., embeddings of individual words): when you are working on a natural language processing task, you are often better off reusing pretrained word embeddings than training your own. The idea of using vectors to represent words dates back to the 1960s, and many sophisticated techniques have been used to generate useful vectors, including using neural networks, but things really took off in 2013, when Tomáš Mikolov and other Google researchers published a paper[10] describing how to learn word embeddings using deep neural networks, much faster than previous attempts. This allowed them to learn embeddings on a very large corpus of text: they trained a deep neural network to predict the words near any given word. This allowed them to obtain astounding word embeddings. For example, synonyms had very close embeddings, and semantically related words such as France, Spain, Italy, and so on, ended up clustered together. But it's not just about proximity: word embeddings were also organized along meaningful axes in the embedding space. Here is a famous example: if you compute King – Man + Woman (adding and subtracting the embedding vectors of these words), then the result will be very close to the embedding of the word Queen (see Figure 13-5). In other words, the word embeddings encode the concept of gender! Similarly, you can compute Madrid – Spain + France, and of course the result is close to Paris, which seems to show that the notion of capital city was also encoded in the embeddings.

10 "Distributed Representations of Words and Phrases and their Compositionality", T. Mikolov et al. (2013).

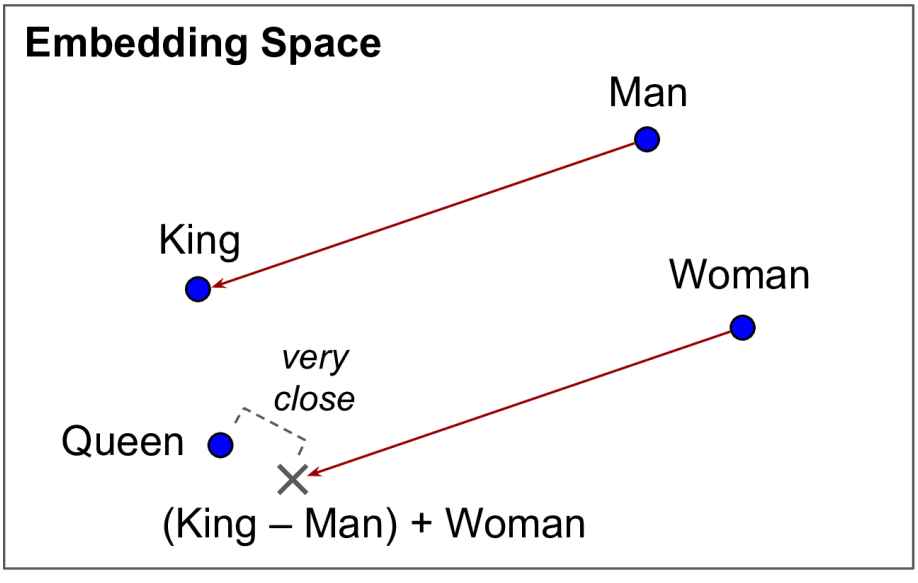

Figure 13-5. Word Embeddings

Let's go back to the Features API. Here is how you could encode the `ocean_proximity` categories as 2D embeddings:

```
ocean_proximity_embed = tf.feature_column.embedding_column(ocean_proximity,
                                                           dimension=2)
```

Each of the five `ocean_proximity` categories will now be represented as a 2D vector. These vectors are stored in an *embedding matrix* with one row per category, and one column per embedding dimension, so in this example it is a 5×2 matrix. When an embedding column is given a category index as input (say, 3, which corresponds to the category `"NEAR BAY"`), it just performs a lookup in the embedding matrix and returns the corresponding row (say, `[0.331, 0.190]`). Unfortunately, the embedding matrix can be quite large, especially when you have a large vocabulary: if this is the case, the model can only learn good representations for the categories for which it has sufficient training data. To reduce the size of the embedding matrix, you can of course try lowering the `dimension` hyperparameter, but if you reduce this parameter too much, the representations may not be as good. Another option is to reduce the vocabulary size (e.g., if you are dealing with text, you can try dropping the rare words from the vocabulary, and replace them all with a token like `"<unknown>"` or `"<UNK>"`). If you are using hash buckets, you can also try reducing the `hash_bucket_size` (but not too much, or else you will get collisions).

If there are no pretrained embeddings that you can reuse for the task you are trying to tackle, and if you do not have enough training data to learn them, then you can try to learn them on some auxiliary task for which it is easier to obtain plenty of training data. After that, you can reuse the trained embeddings for your main task.

Using Feature Columns for Parsing

Let's suppose you have created feature columns for each of your input features, as well as for the target. What can you do with them? Well, for one you can pass them to the `make_parse_example_spec()` function to generate feature descriptions (so you don't have to do it manually, as we did earlier):

```
columns = [bucketized_age, ....., median_house_value] # all features + target
feature_descriptions = tf.feature_column.make_parse_example_spec(columns)
```

You don't always have to create a separate feature column for each and every feature. For example, instead of having 2 numerical feature columns, you could choose to have a single 2D column: just set `shape=[2]` when calling `numerical_column()`.

You can then create a function that parses serialized examples using these feature descriptions, and separates the target column from the input features:

```
def parse_examples(serialized_examples):
    examples = tf.io.parse_example(serialized_examples, feature_descriptions)
    targets = examples.pop("median_house_value") # separate the targets
    return examples, targets
```

Next, you can create a `TFRecordDataset` that will read batches of serialized examples (assuming the TFRecord file contains serialized `Example` protobufs with the appropriate features):

```
batch_size = 32
dataset = tf.data.TFRecordDataset(["my_data_with_features.tfrecords"])
dataset = dataset.repeat().shuffle(10000).batch(batch_size).map(parse_examples)
```

Using Feature Columns in Your Models

Feature columns can also be used directly in your model, to convert all your input features into a single dense vector which the neural network can then process. For this, all you need to do is add a `keras.layers.DenseFeatures` layer as the first layer in your model, passing it the list of feature columns (excluding the target column):

```
columns_without_target = columns[:-1]
model = keras.models.Sequential([
    keras.layers.DenseFeatures(feature_columns=columns_without_target),
```

```
    keras.layers.Dense(1)
])
model.compile(loss="mse", optimizer="sgd", metrics=["accuracy"])
steps_per_epoch = len(X_train) // batch_size
history = model.fit(dataset, steps_per_epoch=steps_per_epoch, epochs=5)
```

The `DenseFeatures` layer will take care of converting every input feature to a dense representation, and it will also apply any extra transformation we specified, such as scaling the `housing_median_age` using the `normalizer_fn` function we provided. You can take a closer look at what the `DenseFeatures` layer does by calling it directly:

```
>>> some_columns = [ocean_proximity_embed, bucketized_income]
>>> dense_features = keras.layers.DenseFeatures(some_columns)
>>> dense_features({
...     "ocean_proximity": [["NEAR OCEAN"], ["INLAND"], ["INLAND"]],
...     "median_income": [[3.], [7.2], [1.]]
... })
...
<tf.Tensor: id=559790, shape=(3, 7), dtype=float32, numpy=
array([[ 0. , 0. , 1. , 0. , 0. ,-0.36277947 , 0.30109018],
       [ 0. , 0. , 0. , 0. , 1. , 0.22548223 , 0.33142096],
       [ 1. , 0. , 0. , 0. , 0. , 0.22548223 , 0.33142096]], dtype=float32)>
```

In this example, we create a `DenseFeatures` layer with just two columns, and we call it with some data, in the form of a dictionary of features. In this case, since the `bucketized_income` column relies on the `median_income` column, the dictionary must include the `"median_income"` key, and similarly since the `ocean_proximity_embed` column is based on the `ocean_proximity` column, the dictionary must include the `"ocean_proximity"` key. Columns are handled in alphabetical order, so first we look at the bucketized income column (its name is the same as the `median_income` column name, plus `"_bucketized"`). The incomes 3, 7.2 and 1 get mapped respectively to category 2 (for incomes between 1.5 and 3), category 0 (for incomes below 1.5), and category 4 (for incomes greater than 6). Then these category IDs get one-hot encoded: category 2 gets encoded as `[0., 0., 1., 0., 0.]` and so on (note that bucketized columns get one-hot encoded by default, no need to call `indicator_column()`). Now on to the `ocean_proximity_embed` column. The `"NEAR OCEAN"` and `"INLAND"` categories just get mapped to their respective embeddings (which were initialized randomly). The resulting tensor is the concatenation of the one-hot vectors and the embeddings.

Now you can feed all kinds of features to a neural network, including numerical features, categorical features, and even text (by splitting the text into words, then using word embedding)! However, performing all the preprocessing on the fly can slow down training. Let's see how this can be improved.

TF Transform

If preprocessing is computationally expensive, then handling it before training rather than on the fly may give you a significant speedup: the data will be preprocessed just once per instance *before* training, rather than once per instance and per epoch *during* training. Tools like Apache Beam let you run efficient data processing pipelines over large amounts of data, even distributed across multiple servers, so why not use it to preprocess all the training data? This works great and indeed can speed up training, but there is one problem: once your model is trained, suppose you want to deploy it to a mobile app: you will need to write some code in your app to take care of preprocessing the data before it is fed to the model. And suppose you also want to deploy the model to TensorFlow.js so it runs in a web browser? Once again, you will need to write some preprocessing code. This can become a maintenance nightmare: whenever you want to change the preprocessing logic, you will need to update your Apache Beam code, your mobile app code and your Javascript code. It is not only time consuming, but also error prone: you may end up with subtle differences between the preprocessing operations performed before training and the ones performed in your app or in the browser. This *training/serving skew* will lead to bugs or degraded performance.

One improvement would be to take the trained model (trained on data that was preprocessed by your Apache Beam code), and before deploying it to your app or the browser, add an extra input layer to take care of preprocessing on the fly (either by writing a custom layer or by using a `DenseFeatures` layer). That's definitely better, since now you just have two versions of your preprocessing code: the Apache Beam code and the preprocessing layer's code.

But what if you could define your preprocessing operations just once? This is what TF Transform was designed for. It is part of TensorFlow Extended (TFX), an end-to-end platform for productionizing TensorFlow models. First, to use a TFX component, such as TF Transform, you must install it, it does not come bundled with TensorFlow. You define your preprocessing function just once (in Python), by using TF Transform functions for scaling, bucketizing, crossing features, and more. You can also use any TensorFlow operation you need. Here is what this preprocessing function might look like if we just had two features:

```
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
    }
```

Next, TF Transform lets you apply this `preprocess()` function to the whole training set using Apache Beam (it provides an `AnalyzeAndTransformDataset` class that you can use for this purpose in your Apache Beam pipeline). In the process, it will also compute all the necessary statistics over the whole training set: in this example, the mean and standard deviation of the `housing_median_age` feature, and the vocabulary for the `ocean_proximity` feature. The components that compute these statistics are called *analyzers*.

Importantly, TF Transform will also generate an equivalent TensorFlow Function that you can plug into the model you deploy. This TF Function contains all the necessary statistics computed by Apache Beam (the mean, standard deviation, and vocabulary), simply included as constants.

At the time of this writing, TF Transform only supports TensorFlow 1. Moreover, Apache Beam only has partial support for Python 3. That said, both these limitations will likely be fixed by the time your read this.

With the Data API, TFRecords, the Features API and TF Transform, you can build highly scalable input pipelines for training, and also benefit from fast and portable data preprocessing in production.

But what if you just wanted to use a standard dataset? Well in that case, things are much simpler: just use TFDS!

The TensorFlow Datasets (TFDS) Project

The TensorFlow Datasets project makes it trivial to download common datasets, from small ones like MNIST or Fashion MNIST, to huge datasets like ImageNet[11] (you will need quite a bit of disk space!). The list includes image datasets, text datasets (including translation datasets), audio and video datasets, and more. You can visit *https://homl.info/tfds* to view the full list, along with a description of each dataset.

TFDS is not bundled with TensorFlow, so you need to install the `tensorflow-datasets` library (e.g., using pip). Then all you need to do is call the `tfds.load()` function, and it will download the data you want (unless it was already downloaded earlier), and return the data as a dictionary of `Datasets` (typically one for training,

11 At the time of writing, TFDS requires you to download a few files manually for ImageNet (for legal reasons), but this will hopefully get resolved soon.

and one for testing, but this depends on the dataset you choose). For example, let's download MNIST:

```
import tensorflow_datasets as tfds

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

You can then apply any transformation you want (typically repeating, batching and prefetching), and you're ready to train your model. Here is a simple example:

```
mnist_train = mnist_train.repeat(5).batch(32).prefetch(1)
for item in mnist_train:
    images = item["image"]
    labels = item["label"]
    [...]
```

In general, `load()` returns a shuffled training set, so there's no need to shuffle it some more.

Note that each item in the dataset is a dictionary containing both the features and the labels. But Keras expects each item to be a tuple containing 2 elements (again, the features and the labels). You could transform the dataset using the `map()` method, like this:

```
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)
```

Or you can just ask the `load()` function to do this for you by setting `as_supervised=True` (obviously this works only for labeled datasets). You can also specify the batch size if you want. Then the dataset can be passed directly to your tf.keras model:

```
dataset = tfds.load(name="mnist", batch_size=32, as_supervised=True)
mnist_train = dataset["train"].repeat().prefetch(1)
model = keras.models.Sequential([...])
model.compile(loss="sparse_categorical_crossentropy", optimizer="sgd")
model.fit(mnist_train, steps_per_epoch=60000 // 32, epochs=5)
```

This was quite a technical chapter, and you may feel that it is a bit far from the abstract beauty of neural networks, but the fact is deep learning often involves large amounts of data, and knowing how to load, parse and preprocess it efficiently is a crucial skill to have. In the next chapter, we will look at Convolutional Neural Networks, which are among the most successful neural net architectures for image processing, and many other applications.