```
...
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)
```

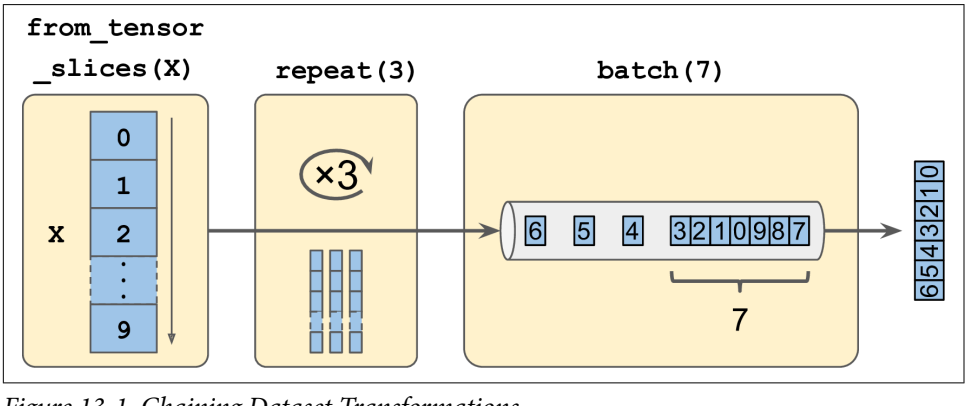

Figure 13-1. Chaining Dataset Transformations

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)
```

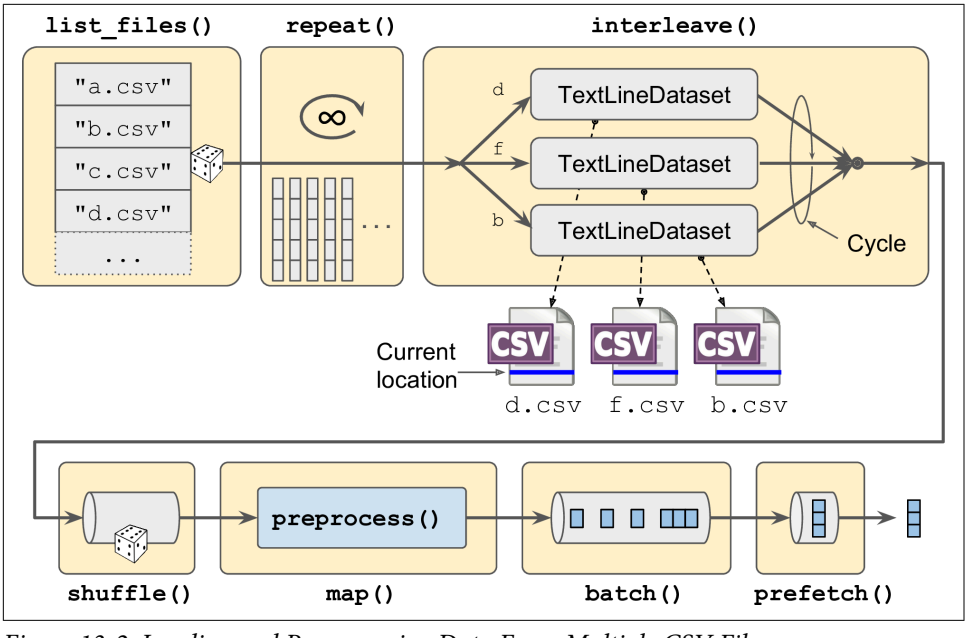

Figure 13-2. Loading and Preprocessing Data From Multiple CSV Files

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).

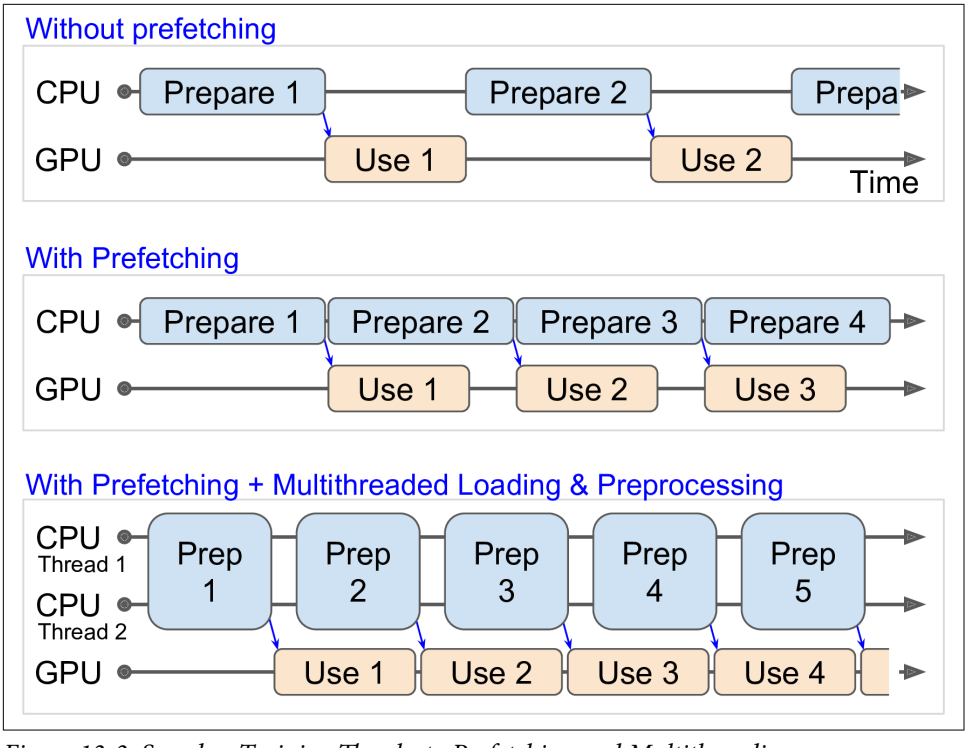

Figure 13-3. Speedup Training Thanks to Prefetching and Multithreading

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.