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 ???).

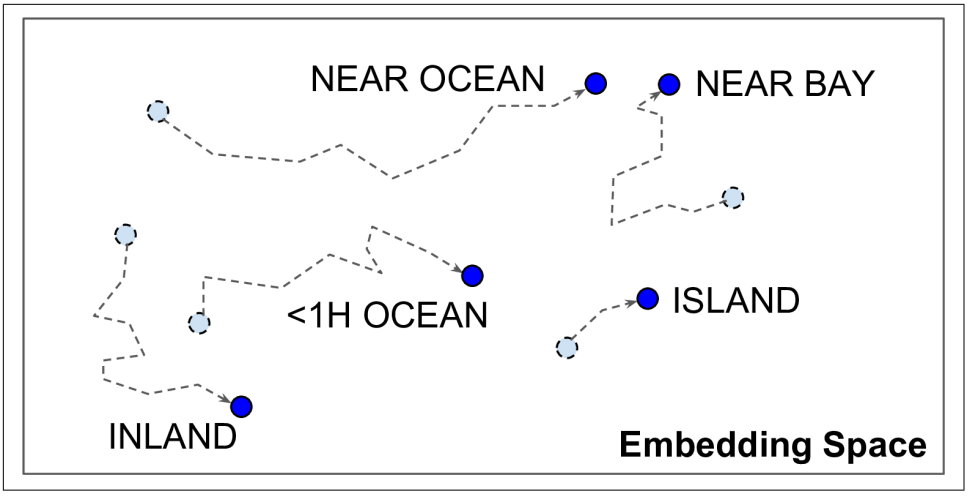

Figure 13-4. Embeddings Will Gradually Improve During Training

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).