03. Loading and Preprocessing Data

In this notebook you will learn how to use TensorFlow's Data API to load and preprocess data efficiently, then you will learn about the efficient `TFRecord` binary format for storing your data.

Imports

In [1]:

```
%matplotlib inline
```

In [2]:

```
import matplotlib as mpl
import matplotlib.pyplot as plt
import numpy as np
import os
import pandas as pd
import sklearn
import sys
import tensorflow as tf
from tensorflow import keras
import time
```

In [3]:

```
print("python", sys.version)
for module in mpl, np, pd, sklearn, tf, keras:
    print(module.__name__, module.__version__)
```

```
python 3.6.8 |Anaconda, Inc.| (default, Dec 30 2018, 01:22:34)
[GCC 7.3.0]
matplotlib 3.0.2
numpy 1.15.4
pandas 0.24.0
sklearn 0.20.2
tensorflow 2.0.0-dev20190124
tensorflow.python.keras.api._v2.keras 2.2.4-tf
```

In [4]:

```
assert sys.version_info >= (3, 5) # Python ≥3.5 required
assert tf.__version__ >= "2.0"    # TensorFlow ≥2.0 required
```

Code examples

You can browse through the code examples or jump directly to the exercises.

In [5]:

```
dataset = tf.data.Dataset.from_tensor_slices(np.arange(10))
dataset
```

Out[5]:

```
<TensorSliceDataset shapes: (), types: tf.int64>
```

In [6]:

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

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

In [7]:

```
dataset = dataset.repeat(3).batch(7)
```

In [8]:

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

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

In [9]:

```
dataset = dataset.interleave(
    lambda v: tf.data.Dataset.from_tensor_slices(v),
    cycle_length=3,
    block_length=2)
```

In [10]:

```
for item in dataset:
    print(item.numpy(), end=" ")
```

```
0 1 7 8 4 5 2 3 9 0 6 7 4 5 1 2 8 9 6 3 0 1 2 8 9 3 4 5 6 7
```

In [11]:

```
X = np.array([[2, 3], [4, 5], [6, 7]])
y = np.array(["cat", "dog", "fox"])
dataset = tf.data.Dataset.from_tensor_slices((X, y))
dataset
```

Out[11]:

```
<TensorSliceDataset shapes: ((2,), ()), types: (tf.int64, tf.string)>
```

In [12]:

```
for item_x, item_y in dataset:
    print(item_x.numpy(), item_y.numpy())
```

```
[2 3] b'cat'
[4 5] b'dog'
[6 7] b'fox'
```

In [13]:

```
dataset = tf.data.Dataset.from_tensor_slices({"features": X, "label": y})
dataset
```

Out[13]:

```
<TensorSliceDataset shapes: {features: (2,), label: ()}, types: {features: tf.
int64, label: tf.string}>
```

In [14]:

```
for item in dataset:
    print(item["features"].numpy(), item["label"].numpy())
```

```
[2 3] b'cat'
[4 5] b'dog'
[6 7] b'fox'
```

Split the California dataset to multiple CSV files

Let's start by loading and preparing the California housing dataset. We first load it, then split it into a training set, a validation set and a test set, and finally we scale it:

In [15]:

```
from sklearn.datasets import fetch_california_housing
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

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

scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_valid_scaled = scaler.transform(X_valid)
X_test_scaled = scaler.transform(X_test)
```

For very large datasets that do not fit in memory, you will typically want to split it into many files first, then have TensorFlow read these files in parallel. To demonstrate this, let's start by splitting the scaled housing dataset and saving it to 20 CSV files:

In [16]:

```
def save_to_multiple_csv_files(data, name_prefix, header=None, n_parts=10):
    housing_dir = os.path.join("datasets", "housing")
    os.makedirs(housing_dir, exist_ok=True)
    path_format = os.path.join(housing_dir, "my_{}_{:02d}.csv")

    filenames = []
    m = len(data)
    for file_idx, row_indices in enumerate(np.array_split(np.arange(m), n_part
s)):
        part_csv = path_format.format(name_prefix, file_idx)
        filenames.append(part_csv)
        with open(part_csv, "wt", encoding="utf-8") as f:
            if header is not None:
                f.write(header)
                f.write("\n")
            for row_idx in row_indices:
                f.write(",".join([repr(col) for col in data[row_idx]]))
                f.write("\n")
    return filenames
```

In [17]:

```
train_data = np.c_[X_train_scaled, y_train]
valid_data = np.c_[X_valid_scaled, y_valid]
test_data = np.c_[X_test_scaled, y_test]
header_cols = ["Scaled" + name for name in housing.feature_names] + ["MedianHouseValue"]
header = ",".join(header_cols)

train_filenames = save_to_multiple_csv_files(train_data, "train", header, n_parts=20)
valid_filenames = save_to_multiple_csv_files(valid_data, "valid", header, n_parts=10)
test_filenames = save_to_multiple_csv_files(test_data, "test", header, n_parts=10)
```

Okay, now let's take a peek at the first few lines of one of these CSV files:

In [18]:

```
with open(train_filenames[0]) as f:
    for i in range(3):
        print(f.readline(), end="")
```

```
ScaledMedInc,ScaledHouseAge,ScaledAveRooms,ScaledAveBedrms,ScaledPopulation,Sc
aledAveOccup,ScaledLatitude,ScaledLongitude,MedianHouseValue
-0.1939788334343112,-1.0778131900560315,-0.9433854492905827,0.0148531378478594
4,0.020733351231179677,-0.5729162417603235,0.9292604730832086,-1.4221552292446
311,1.442
0.7519831792363448,-1.8688949973875395,0.4054779316683507,-0.2332768194594582,
1.861464900604635,0.20516532460775205,-0.9165473773933427,1.096669692658571,1.
687
```

Exercise 1 – Data API

1.1)

Use `tf.data.Dataset.list_files()` to create a dataset that will simply list the training filenames. Iterate through its items and print them.

In [19]:

```
filename_dataset = tf.data.Dataset.list_files(train_filenames)
```

In [20]:

```
for filename in filename_dataset:
    print(filename)
```

```
tf.Tensor(b'datasets/housing/my_train_16.csv', shape=(), dtype=string)
tf.Tensor(b'datasets/housing/my_train_01.csv', shape=(), dtype=string)
tf.Tensor(b'datasets/housing/my_train_00.csv', shape=(), dtype=string)
tf.Tensor(b'datasets/housing/my_train_17.csv', shape=(), dtype=string)
tf.Tensor(b'datasets/housing/my_train_09.csv', shape=(), dtype=string)
tf.Tensor(b'datasets/housing/my_train_12.csv', shape=(), dtype=string)
tf.Tensor(b'datasets/housing/my_train_04.csv', shape=(), dtype=string)
tf.Tensor(b'datasets/housing/my_train_02.csv', shape=(), dtype=string)
tf.Tensor(b'datasets/housing/my_train_03.csv', shape=(), dtype=string)
tf.Tensor(b'datasets/housing/my_train_08.csv', shape=(), dtype=string)
tf.Tensor(b'datasets/housing/my_train_18.csv', shape=(), dtype=string)
tf.Tensor(b'datasets/housing/my_train_06.csv', shape=(), dtype=string)
tf.Tensor(b'datasets/housing/my_train_07.csv', shape=(), dtype=string)
tf.Tensor(b'datasets/housing/my_train_15.csv', shape=(), dtype=string)
tf.Tensor(b'datasets/housing/my_train_11.csv', shape=(), dtype=string)
tf.Tensor(b'datasets/housing/my_train_19.csv', shape=(), dtype=string)
tf.Tensor(b'datasets/housing/my_train_10.csv', shape=(), dtype=string)
tf.Tensor(b'datasets/housing/my_train_05.csv', shape=(), dtype=string)
tf.Tensor(b'datasets/housing/my_train_13.csv', shape=(), dtype=string)
tf.Tensor(b'datasets/housing/my_train_14.csv', shape=(), dtype=string)
```

1.2)

Use the filename dataset's `interleave()` method to create a dataset that will read from these CSV files, interleaving their lines. The first argument needs to be a function (e.g., a `lambda`) that creates a `tf.data.TextLineDataset` based on a filename, and you must also set `cycle_length=5` so that the reader interleaves data from 5 files at a time. Print the first 15 elements from this dataset to see that you do indeed get interleaved lines from multiple CSV files (you should get the first line from 5 files, then the second line from these same files, then the third lines). **Tip**: To get only the first 15 elements, you can call the dataset's `take()` method.

In [21]:

```
n_readers = 5
dataset = filename_dataset.interleave(
    lambda filename: tf.data.TextLineDataset(filename),
    cycle_length=n_readers)
```

In [22]:
```
for line in dataset.take(15):
    print(line.numpy())
```

```
b'ScaledMedInc,ScaledHouseAge,ScaledAveRooms,ScaledAveBedrms,ScaledPopulation,
ScaledAveOccup,ScaledLatitude,ScaledLongitude,MedianHouseValue'
b'ScaledMedInc,ScaledHouseAge,ScaledAveRooms,ScaledAveBedrms,ScaledPopulation,
ScaledAveOccup,ScaledLatitude,ScaledLongitude,MedianHouseValue'
b'ScaledMedInc,ScaledHouseAge,ScaledAveRooms,ScaledAveBedrms,ScaledPopulation,
ScaledAveOccup,ScaledLatitude,ScaledLongitude,MedianHouseValue'
b'ScaledMedInc,ScaledHouseAge,ScaledAveRooms,ScaledAveBedrms,ScaledPopulation,
ScaledAveOccup,ScaledLatitude,ScaledLongitude,MedianHouseValue'
b'ScaledMedInc,ScaledHouseAge,ScaledAveRooms,ScaledAveBedrms,ScaledPopulation,
ScaledAveOccup,ScaledLatitude,ScaledLongitude,MedianHouseValue'
b'0.1515976772572887З,1.849189497070548,0.09624145072689123,-0.221516199133773
74,-0.6682861429567027,-0.23549309239259542,-0.8978082113986573,0.636884191041
3197,3.215'
b'-0.10698238234932358,1.2163240512053415,-0.36522430755132806,-0.229080294438
43958,-0.882749164326441,0.10033860694954831,-0.7525796749398415,0.71684688697
47529,1.625'
b'-0.1939788334343112,-1.0778131900560315,-0.9433854492905827,0.01485313784785
944,0.020733351231179677,-0.5729162417603235,0.9292604730832086,-1.42215522924
46311,1.442'
b'-0.28605575156038454,0.662566786073286,-0.3692986080768526,-0.41074201720561
043,-0.8818365557248676,0.11285159127079294,0.4795204892107495,-1.057325429048
3325,2.526'
b'-0.024176007174509982,0.5043504246069844,0.00212128454489414,-0.305136235257
42406,-0.2466609690298131,-0.03432783615893503,1.0838585925393653,-1.232243826
4027185,1.598'
b'0.2910224531683286,-0.28673138272452353,0.347728342329676,-0.16536721847336З
64,-0.45291051298539114,-0.13973664455937504,1.1166521330300656,-1.33719486481
53558,2.037'
b'0.052345254135042395,-0.28673138272452353,0.047219258690479195,-0.1398385763
7224692,0.16218768447504958,-0.019539119643793 85,1.2571958779902077,-1.6170643
00582372,2.441'
b'0.7519831792363448,-1.8688949973875395,0.4054779316683507,-0.233276819459458
2,1.861464900604635,0.20516532460775205,-0.9165473773933427,1.09666969265857 1,
1.687'
b'-0.8687905587678326,-0.28673138272452353,-0.27797755761223253,0.070672829405
65517,-1.158356962001593 9,-0.46218586965369224,1.3743156654569937,-0.817437341
2480258,0.55'
b'0.41326791784765465,-0.3658395634576743,-0.07742256595420528,-0.158775666485
84598,0.45969808858796307,-0.298086570677949,-0.9352865433880281,0.84178859937
07449,2.455'
```

1.3)

We do not care about the header lines, so let's skip them. You can use the `skip()` method for this. Print the first five elements of your final dataset to make sure it does not print any header lines. **Tip**: make sure to call `skip()` for each `TextLineDataset`, not for the interleave dataset.

In [23]:
```
dataset = filename_dataset.interleave(
    lambda filename: tf.data.TextLineDataset(filename).skip(1),
    cycle_length=n_readers)
```

```
In [24]: for line in dataset.take(5):
             print(line.numpy())
```

```
b'0.15159767725728873,1.849189497070548,0.09624145072689123,-0.22151619913377374,-0.6682861429567027,-0.23549309239259542,-0.8978082113986573,0.6368841910413197,3.215'
b'-0.10698238234932358,1.2163240512053415,-0.36522430755132806,-0.22908029443843958,-0.882749164326441,0.10033860694954831,-0.7525796749398415,0.7168468869747529,1.625'
b'-0.1939788334343112,-1.0778131900560315,-0.9433854492905827,0.01485313784785944,0.020733351231179677,-0.5729162417603235,0.9292604730832086,-1.4221552292446311,1.442'
b'-0.28605575156038454,0.662566786073286,-0.3692986080768526,-0.41074201720561043,-0.8818365557248676,0.11285159127079294,0.4795204892107495,-1.0573254290483325,2.526'
b'-0.024176007174509982,0.5043504246069844,0.00212128454489414,-0.30513623525742406,-0.2466609690298131,-0.03432783615893503,1.0838585925393653,-1.2322438264027185,1.598'
```

1.4)

We need to parse these CSV lines. First, experiment with the `tf.io.decode_csv()` function using the example below (e.g., look at the types, try changing or removing some field values, etc.).

- You need to pass it the line to parse, and set the `record_defaults` argument. This must be an array containing the default value for each field, in case it is missing. This also tells TensorFlow the number of fields to expect, and the type of each field. If you do not want a default value for a given field, you must use an empty tensor of the appropriate type (e.g., `tf.constant([])` for a `float32` field, or `tf.constant([], dtype=tf.int64` for an `int64` field).

```
In [25]: record_defaults=[0, np.nan, tf.constant(np.nan, dtype=tf.float64), "Hello", tf
         .constant([])]
         parsed_fields = tf.io.decode_csv('1,2,3,4,5', record_defaults)
         parsed_fields
```

```
Out[25]: [<tf.Tensor: id=303, shape=(), dtype=int32, numpy=1>,
          <tf.Tensor: id=304, shape=(), dtype=float32, numpy=2.0>,
          <tf.Tensor: id=305, shape=(), dtype=float64, numpy=3.0>,
          <tf.Tensor: id=306, shape=(), dtype=string, numpy=b'4'>,
          <tf.Tensor: id=307, shape=(), dtype=float32, numpy=5.0>]
```

Notice that all missing fields are replaced with their default value, when provided:

```
In [26]: parsed_fields = tf.io.decode_csv(',,,,5', record_defaults)
         parsed_fields
```

```
Out[26]: [<tf.Tensor: id=317, shape=(), dtype=int32, numpy=0>,
          <tf.Tensor: id=318, shape=(), dtype=float32, numpy=nan>,
          <tf.Tensor: id=319, shape=(), dtype=float64, numpy=nan>,
          <tf.Tensor: id=320, shape=(), dtype=string, numpy=b'Hello'>,
          <tf.Tensor: id=321, shape=(), dtype=float32, numpy=5.0>]
```

The 5th field is compulsory (since we provided `tf.constant([])` as the "default value"), so we get an exception if we do not provide it:

In [27]:
```
try:
    parsed_fields = tf.io.decode_csv(',,,,', record_defaults)
except tf.errors.InvalidArgumentError as ex:
    print(ex)
```

```
Field 4 is required but missing in record 0! [Op:DecodeCSV]
```

The number of fields should match exactly the number of fields in the `record_defaults`:

In [28]:
```
try:
    parsed_fields = tf.io.decode_csv('1,2,3,4,5,6,7', record_defaults)
except tf.errors.InvalidArgumentError as ex:
    print(ex)
```

```
Expect 5 fields but have 7 in record 0 [Op:DecodeCSV]
```

1.5)

Now you are ready to create a function to parse a CSV line:

- Create a `parse_csv_line()` function that takes a single line as argument.
- Call `tf.io.decode_csv()` to parse that line.
- Call `tf.stack()` to create a single tensor containing all the input features (i.e., all fields except the last one).
- Reshape the labels field (i.e., the last field) to give it a shape of `[1]` instead of `[]` (i.e., it must not be a scalar). You can use `tf.reshape(label_field, [1])`, or call `tf.stack([label_field])`, or use `label_field[tf.newaxis]`.
- Return a tuple with both tensors (input features and labels).
- Try calling it on a single line from one of the CSV files.

In [29]:
```
n_inputs = X_train.shape[1]

def parse_csv_line(line, n_inputs=n_inputs):
    defs = [tf.constant(np.nan)] * (n_inputs + 1)
    fields = tf.io.decode_csv(line, record_defaults=defs)
    x = tf.stack(fields[:-1])
    y = tf.stack(fields[-1:])
    return x, y
```

In [30]:
```
parse_csv_line(b'-0.739840972632228,-0.3658395634576743,-0.784679995482575,0.0
7414513752253027,0.7544706668961565,0.407700592469922,-0.686992593958441,0.601
9005115704453,2.0')
```

Out[30]:
```
(<tf.Tensor: id=346, shape=(8,), dtype=float32, numpy=
 array([-0.739841  , -0.36583957, -0.78468   ,  0.07414514,  0.75447065,
         0.4077006 , -0.6869926 ,  0.6019005 ], dtype=float32)>,
 <tf.Tensor: id=347, shape=(1,), dtype=float32, numpy=array([2.], dtype=float3
2)>)
```

1.6)

Now create a `csv_reader_dataset()` function that takes a list of CSV filenames and returns a dataset that will provide batches of parsed and shuffled data from these files, including the features and labels, repeating the whole data once per epoch.

Tips:

- Copy your code from above to get a dataset that returns interleaved lines from the given CSV files. Your function will need an argument for the `filenames`, and another for the number of files read in parallel at any given time (e.g., `n_reader`).
- The training algorithm will need to go through the dataset many times, so you should call `repeat()` on the filenames dataset. You do not need to specify a number of repetitions, as we will tell Keras the number of iterations to run later on.
- Gradient descent works best when the data is IID (independent and identically distributed), so you should call the `shuffle()` method. It will require the shuffling buffer size, which you can add as an argument to your function (e.g., `shuffle_buffer_size`).
- Use the `map()` method to apply the `parse_csv_line()` function to each CSV line. You can set the `num_parallel_calls` argument to the number of threads that will parse lines in parallel. This should probably be an argument of your function (e.g., `n_parse_threads`).
- Use the `batch()` method to bundle records into batches. You will need to specify the batch size. This should probably be an argument of your function (e.g., `batch_size`).
- Call `prefetch(1)` on your final dataset to ensure that the next batch is loaded and parsed while the rest of your computations take place in parallel (to avoid blocking for I/O).
- Return the resulting dataset.
- Give every argument a reasonable default value (except for the filenames).
- Test your function by calling it with a small batch size and printing the first couple of batches.
- For higher performance, you can replace `dataset.map(...).batch(...)` with `dataset.apply(map_and_batch(...))`, where `map_and_batch()` is an experimental function located in `tf.data.experimental`. It will be deprecated in future versions of TensorFlow when such pipeline optimizations become automatic.

In [31]:

```python
def csv_reader_dataset(filenames, n_parse_threads=5, batch_size=32,
                       shuffle_buffer_size=10000, n_readers=5):
    dataset = tf.data.Dataset.list_files(filenames)
    dataset = dataset.repeat()
    dataset = dataset.interleave(
        lambda filename: tf.data.TextLineDataset(filename).skip(1),
        cycle_length=n_readers)
    dataset.shuffle(shuffle_buffer_size)
    dataset = dataset.map(parse_csv_line, num_parallel_calls=n_parse_threads)
    dataset = dataset.batch(batch_size)
    return dataset.prefetch(1)
```

This version uses `map_and_batch()` to get a performance boost (but remember that this feature is experimental and will eventually be deprecated, as explained earlier):

In [32]:

```
def csv_reader_dataset(filenames, batch_size=32,
                       shuffle_buffer_size=10000, n_readers=5):
    dataset = tf.data.Dataset.list_files(filenames)
    dataset = dataset.repeat()
    dataset = dataset.interleave(
        lambda filename: tf.data.TextLineDataset(filename).skip(1),
        cycle_length=n_readers)
    dataset.shuffle(shuffle_buffer_size)
    dataset = dataset.apply(
        tf.data.experimental.map_and_batch(
            parse_csv_line,
            batch_size,
            num_parallel_calls=tf.data.experimental.AUTOTUNE))
    return dataset.prefetch(1)
```

In [33]:

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

```
X = tf.Tensor(
[[-0.19397883 -1.0778131  -0.9433854   0.01485314  0.02073335 -0.57291627
   0.9292605  -1.4221553 ]
 [ 2.5288372   1.2163241   0.27730894 -0.11517556 -0.69018877 -0.06940325
  -0.67762303  0.7018539 ]
 [-0.11923835 -0.91959685  0.04752479  0.11741281 -0.8334683  -0.01645926
   2.4377635  -0.7424723 ]], shape=(3, 8), dtype=float32)
y = tf.Tensor(
[[1.442]
 [4.159]
 [0.808]], shape=(3, 1), dtype=float32)

X = tf.Tensor(
[[-0.45570144 -0.5240559  -0.18509613  0.00251733 -0.738557   -0.20285924
   1.4586419  -0.5075819 ]
 [-0.02417601  0.5043504   0.00212128 -0.30513623 -0.24666096 -0.03432783
   1.0838586  -1.2322438 ]
 [ 0.75198317 -1.868895    0.40547794 -0.23327681  1.8614649   0.20516533
  -0.91654736  1.0966697 ]], shape=(3, 8), dtype=float32)
y = tf.Tensor(
[[1.069]
 [1.598]
 [1.687]], shape=(3, 1), dtype=float32)
```

1.7)

Build a training set, a validation set and a test set using your `csv_reader_dataset()` function.

In [34]:

```
batch_size = 32
train_set = csv_reader_dataset(train_filenames, batch_size)
valid_set = csv_reader_dataset(valid_filenames, batch_size)
test_set = csv_reader_dataset(test_filenames, batch_size)
```

1.8)

Build and compile a Keras model for this regression task, and use your datasets to train it, evaluate it and make predictions for the test set.

Tips

- Instead of passing `X train_scaled, y_train` to the `fit()` method, pass the training dataset and specify the `steps_per_epoch` argument. This should be set to the number of instances in the training set divided by the batch size.
- Similarly, pass the validation dataset instead of `(X_valid_scaled, y_valid)` and `y_valid`, and set the `validation steps`.
- For the `evaluate()` and `predict()` methods, you need to pass the test dataset, and specify the `steps` argument.
- The `predict()` method ignores the labels in the test dataset, but if you want to be extra sure that it does not cheat, you can create a new dataset by stripping away the labels from the test set (e.g., `test_set.map(lambda X, y: X)`).

In [35]:

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

In [36]:

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

In [37]:

```
model.fit(train_set, steps_per_epoch=len(X_train) // batch_size, epochs=10,
          validation_data=valid_set, validation_steps=len(X_valid) // batch_si
ze)
```

```
WARNING: Logging before flag parsing goes to stderr.
W0219 08:54:43.408062 140432385967872 deprecation.py:323] From /opt/anaconda3/
lib/python3.6/site-packages/tensorflow/python/data/ops/dataset_ops.py:1730: Da
tasetV1.make_one_shot_iterator (from tensorflow.python.data.ops.dataset_ops) i
s deprecated and will be removed in a future version.
Instructions for updating:
Use `for ... in dataset:` to iterate over a dataset. If using `tf.estimator`,
return the `Dataset` object directly from your input function. As a last resor
t, you can use `tf.compat.v1.data.make_one_shot_iterator(dataset)`.
```

```
Epoch 1/10
362/362==============================] - 2s 6ms/step - loss: 2.1300 - val_los
s: 3.5107
Epoch 2/10
362/362==============================] - 1s 3ms/step - loss: 0.8448 - val_los
s: 0.8774
Epoch 3/10
362/362==============================] - 1s 3ms/step - loss: 0.7268 - val_los
s: 0.6817
Epoch 4/10
362/362==============================] - 1s 3ms/step - loss: 0.6810 - val_los
s: 0.6327
Epoch 5/10
362/362==============================] - 1s 3ms/step - loss: 0.6556 - val_los
s: 0.6040
Epoch 6/10
362/362==============================] - 1s 3ms/step - loss: 0.6249 - val_los
s: 0.5868
Epoch 7/10
362/362==============================] - 1s 3ms/step - loss: 0.5965 - val_los
s: 0.5750
Epoch 8/10
362/362==============================] - 1s 3ms/step - loss: 0.5791 - val_los
s: 0.5616
Epoch 9/10
362/362==============================] - 1s 3ms/step - loss: 0.5588 - val_los
s: 0.5313
Epoch 10/10
362/362==============================] - 1s 3ms/step - loss: 0.5375 - val_los
s: 0.5005
```

Out[37]:

```
<tensorflow.python.keras.callbacks.History at 0x7fb8601d6518>
```

In [38]:

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

```
161/161==============================] - 0s 2ms/step - loss: 0.5278
```

Out[38]:

```
0.5278426802121334
```

In [39]:

```
new_set = test_set.map(lambda X, y: X)
model.predict(new_set, steps=len(X_test) // batch_size)
```

Out[39]:

```
array([[2.525579 ],
       [1.1378739],
       [1.7856278],
       ...,
       [2.2110467],
       [1.4341481],
       [2.594335 ]], dtype=float32)
```

Exercise 2 – The `TFRecord` binary format

Code examples

You can walk through these code examples or jump down to the actual exercise below.

In [40]:

```
favorite_books = [name.encode("utf-8")
                  for name in ["Arluk", "Fahrenheit 451", "L'étranger"]]
favorite_books = tf.train.BytesList(value=favorite_books)
favorite_books
```

Out[40]:

```
value: "Arluk"
value: "Fahrenheit 451"
value: "L\'\303\251tranger"
```

In [41]:

```
hours_per_month = tf.train.FloatList(value=[15.5, 9.5, np.nan, 6.0, 9.0])
hours_per_month
```

Out[41]:

```
value: 15.5
value: 9.5
value: nan
value: 6.0
value: 9.0
```

In [42]:

```
age = tf.train.Int64List(value=[42])
age
```

Out[42]:

```
value: 42
```

In [43]:

```
coordinates = tf.train.FloatList(value=[1.2834, 103.8607])
coordinates
```

Out[43]:

```
value: 1.283400058746338
value: 103.86070251464844
```

In [44]:

```
features = tf.train.Features(
    feature={
        "favorite_books": tf.train.Feature(bytes_list=favorite_books),
        "hours_per_month": tf.train.Feature(float_list=hours_per_month),
        "age": tf.train.Feature(int64_list=age),
        "coordinates": tf.train.Feature(float_list=coordinates),
    }
)
features
```

Out[44]:

```
feature {
  key: "age"
  value {
    int64_list {
      value: 42
    }
  }
}
feature {
  key: "coordinates"
  value {
    float_list {
      value: 1.2834000587463379
      value: 103.86070251464844
    }
  }
}
feature {
  key: "favorite_books"
  value {
    bytes_list {
      value: "Arluk"
      value: "Fahrenheit 451"
      value: "L\'\303\251tranger"
    }
  }
}
feature {
  key: "hours_per_month"
  value {
    float_list {
      value: 15.5
      value: 9.5
      value: nan
      value: 6.0
      value: 9.0
    }
  }
}
```

In [45]:

```
example = tf.train.Example(features=features)
example
```

Out[45]:

```
features {
  feature {
    key: "age"
    value {
      int64_list {
        value: 42
      }
    }
  }
  feature {
    key: "coordinates"
    value {
      float_list {
        value: 1.283400058746338
        value: 103.86070251464844
      }
    }
  }
  feature {
    key: "favorite_books"
    value {
      bytes_list {
        value: "Arluk"
        value: "Fahrenheit 451"
        value: "L\'\303\251tranger"
      }
    }
  }
  feature {
    key: "hours_per_month"
    value {
      float_list {
        value: 15.5
        value: 9.5
        value: nan
        value: 6.0
        value: 9.0
      }
    }
  }
}
```

In [46]:

```
serialized_example = example.SerializeToString()
serialized_example
```

Out[46]:

```
b"\n\x92\x01\n\x1b\n\x0bcoordinates\x12\x0c\x12\n\n\x08tF\xa4?\xae\xb8\xcfB\n+
\n\x0fhours_per_month\x12\x18\x12\x16\n\x14\x00\x00xA\x00\x00\x18A\x00\x00\xc0
\x7f\x00\x00\xc0@\x00\x00\x10A\n\x0c\n\x03age\x12\x05\x1a\x03\n\x01*\n8\n\x0ef
avorite_books\x12&\n$\n\x05Arluk\n\x0eFahrenheit 451\n\x0bL'\xc3\xa9tranger"
```

In [47]:

```
filename = "my_reading_data.tfrecords"
with tf.io.TFRecordWriter(filename) as writer:
    for i in range(3): # you should save different examples instead! :)
        writer.write(serialized_example)
```

In [48]:
```
for serialized_example_tensor in tf.data.TFRecordDataset([filename]):
    print(serialized_example_tensor)
```

```
tf.Tensor(b"\n\x92\x01\n\x1b\n\x0bcoordinates\x12\x0c\x12\n\n\x08tF\xa4?\xae\x
b8\xcfB\n+\n\x0fhours_per_month\x12\x18\x12\x16\n\x14\x00\x00xA\x00\x00\x18A\x
00\x00\xc0\x7f\x00\x00\xc0@\x00\x00\x10A\n\x0c\n\x03age\x12\x05\x1a\x03\n\x01*
\n8\n\x0efavorite_books\x12&\n$\n\x05Arluk\n\x0eFahrenheit 451\n\x0bL'\xc3\xa9
tranger", shape=(), dtype=string)
tf.Tensor(b"\n\x92\x01\n\x1b\n\x0bcoordinates\x12\x0c\x12\n\n\x08tF\xa4?\xae\x
b8\xcfB\n+\n\x0fhours_per_month\x12\x18\x12\x16\n\x14\x00\x00xA\x00\x00\x18A\x
00\x00\xc0\x7f\x00\x00\xc0@\x00\x00\x10A\n\x0c\n\x03age\x12\x05\x1a\x03\n\x01*
\n8\n\x0efavorite_books\x12&\n$\n\x05Arluk\n\x0eFahrenheit 451\n\x0bL'\xc3\xa9
tranger", shape=(), dtype=string)
tf.Tensor(b"\n\x92\x01\n\x1b\n\x0bcoordinates\x12\x0c\x12\n\n\x08tF\xa4?\xae\x
b8\xcfB\n+\n\x0fhours_per_month\x12\x18\x12\x16\n\x14\x00\x00xA\x00\x00\x18A\x
00\x00\xc0\x7f\x00\x00\xc0@\x00\x00\x10A\n\x0c\n\x03age\x12\x05\x1a\x03\n\x01*
\n8\n\x0efavorite_books\x12&\n$\n\x05Arluk\n\x0eFahrenheit 451\n\x0bL'\xc3\xa9
tranger", shape=(), dtype=string)
```

In [49]:
```
filename = "my_reading_data.tfrecords"
options = tf.io.TFRecordOptions(compression_type="GZIP")
with tf.io.TFRecordWriter(filename, options) as writer:
    for i in range(3): # you should save different examples instead! :)
        writer.write(serialized_example)
```

In [50]:
```
dataset = tf.data.TFRecordDataset([filename], compression_type="GZIP")
for serialized_example_tensor in dataset:
    print(serialized_example_tensor)
```

```
tf.Tensor(b"\n\x92\x01\n\x1b\n\x0bcoordinates\x12\x0c\x12\n\n\x08tF\xa4?\xae\x
b8\xcfB\n+\n\x0fhours_per_month\x12\x18\x12\x16\n\x14\x00\x00xA\x00\x00\x18A\x
00\x00\xc0\x7f\x00\x00\xc0@\x00\x00\x10A\n\x0c\n\x03age\x12\x05\x1a\x03\n\x01*
\n8\n\x0efavorite_books\x12&\n$\n\x05Arluk\n\x0eFahrenheit 451\n\x0bL'\xc3\xa9
tranger", shape=(), dtype=string)
tf.Tensor(b"\n\x92\x01\n\x1b\n\x0bcoordinates\x12\x0c\x12\n\n\x08tF\xa4?\xae\x
b8\xcfB\n+\n\x0fhours_per_month\x12\x18\x12\x16\n\x14\x00\x00xA\x00\x00\x18A\x
00\x00\xc0\x7f\x00\x00\xc0@\x00\x00\x10A\n\x0c\n\x03age\x12\x05\x1a\x03\n\x01*
\n8\n\x0efavorite_books\x12&\n$\n\x05Arluk\n\x0eFahrenheit 451\n\x0bL'\xc3\xa9
tranger", shape=(), dtype=string)
tf.Tensor(b"\n\x92\x01\n\x1b\n\x0bcoordinates\x12\x0c\x12\n\n\x08tF\xa4?\xae\x
b8\xcfB\n+\n\x0fhours_per_month\x12\x18\x12\x16\n\x14\x00\x00xA\x00\x00\x18A\x
00\x00\xc0\x7f\x00\x00\xc0@\x00\x00\x10A\n\x0c\n\x03age\x12\x05\x1a\x03\n\x01*
\n8\n\x0efavorite_books\x12&\n$\n\x05Arluk\n\x0eFahrenheit 451\n\x0bL'\xc3\xa9
tranger", shape=(), dtype=string)
```

In [51]:

```
expected_features = {
    "favorite_books": tf.io.VarLenFeature(dtype=tf.string),
    "hours_per_month": tf.io.VarLenFeature(dtype=tf.float32),
    "age": tf.io.FixedLenFeature([], dtype=tf.int64),
    "coordinates": tf.io.FixedLenFeature([2], dtype=tf.float32),
}

for serialized_example_tensor in tf.data.TFRecordDataset(
        [filename], compression_type="GZIP"):
    example = tf.io.parse_single_example(serialized_example_tensor,
                                         expected_features)
    books = tf.sparse.to_dense(example["favorite_books"],
                               default_value=b"")
    for book in books:
        print(book.numpy().decode('UTF-8'), end="\t")
    print()
```

```
Arluk   Fahrenheit 451  L'étranger
Arluk   Fahrenheit 451  L'étranger
Arluk   Fahrenheit 451  L'étranger
```

Actual exercise

2.1)

Write a `csv_to_tfrecords()` function that will read from a given CSV dataset (e.g., such as `train_set`, passed as an argument), and write the instances to multiple TFRecord files. The number of files should be defined by an `n_shards` argument. If there are, say, 20 shards, then the files should be named `my_train_00000-to-00019.tfrecords` to `my_train_00019-to-00019.tfrecords`, where the `my_train` prefix should be defined by an argument.

Tips:

- since the CSV dataset repeats the dataset forever, the function should take an argument defining the number of steps per shard, and you should use `take()` to pull only the appropriate number of batches from the CSV dataset for each shard.
- to format 19 as `"00019"`, you can use `"{:05d}".format(19)`.

In [52]:

```
def serialize_example(x, y):
    input_features = tf.train.FloatList(value=x)
    label = tf.train.FloatList(value=y)
    features = tf.train.Features(
        feature = {
            "input_features": tf.train.Feature(float_list=input_features),
            "label": tf.train.Feature(float_list=label),
        }
    )
    example = tf.train.Example(features=features)
    return example.SerializeToString()
```

In [53]:

```python
def csv_to_tfrecords(filename, csv_reader_dataset, n_shards, steps_per_shard,
                     compression_type=None):
    options = tf.io.TFRecordOptions(compression_type=compression_type)
    for shard in range(n_shards):
        path = "{}_{:05d}-of-{:05d}.tfrecords".format(filename, shard, n_shards)
        with tf.io.TFRecordWriter(path, options) as writer:
            for X_batch, y_batch in csv_reader_dataset.take(steps_per_shard):
                for x_instance, y_instance in zip(X_batch, y_batch):
                    writer.write(serialize_example(x_instance, y_instance))
```

2.2)

Use this function to write the training set, validation set and test set to multiple TFRecord files.

In [54]:

```python
batch_size = 32
n_shards = 20
steps_per_shard = len(X_train) // batch_size // n_shards
csv_to_tfrecords("my_train.tfrecords", train_set, n_shards, steps_per_shard)

n_shards = 1
steps_per_shard = len(X_valid) // batch_size // n_shards
csv_to_tfrecords("my_valid.tfrecords", valid_set, n_shards, steps_per_shard)

n_shards = 1
steps_per_shard = len(X_test) // batch_size // n_shards
csv_to_tfrecords("my_test.tfrecords", test_set, n_shards, steps_per_shard)
```

2.3)

Write a `tfrecords_reader_dataset()` function, very similar to `csv_reader_dataset()`, that will read from multiple TFRecord files. For convenience, it should take a file prefix (such as `"my_train"`) and use `os.listdir()` to look for all the TFRecord files with that prefix.

Tips:

- You can mostly reuse `csv_reader_dataset()`, except it will use a different parsing function (based on `tf.io.parse_single_example()` instead of `tf.io.parse_csv_line()`).
- The parsing function should return `(input_features, label)`, not a `tf.train.Example`.

In [55]:

```python
expected_features = {
    "input_features": tf.io.FixedLenFeature([n_inputs], dtype=tf.float32),
    "label": tf.io.FixedLenFeature([1], dtype=tf.float32),
}

def parse_tfrecord(serialized_example):
    example = tf.io.parse_single_example(serialized_example,
                                         expected_features)
    return example["input_features"], example["label"]
```

In [56]:

```
def tfrecords_reader_dataset(filename, batch_size=32,
                             shuffle_buffer_size=10000, n_readers=5):
    filenames = [name for name in os.listdir() if name.startswith(filename)
                                               and name.endswith(".tfrecords")]
    dataset = tf.data.Dataset.list_files(filenames)
    dataset = dataset.repeat()
    dataset = dataset.interleave(
        lambda filename: tf.data.TFRecordDataset(filename),
        cycle_length=n_readers)
    dataset.shuffle(shuffle_buffer_size)
    dataset = dataset.apply(
        tf.data.experimental.map_and_batch(
            parse_tfrecord,
            batch_size,
            num_parallel_calls=tf.data.experimental.AUTOTUNE))
    return dataset.prefetch(1)
```

In [57]:

```
tfrecords_train_set = tfrecords_reader_dataset("my_train", batch_size=3)
for X_batch, y_batch in tfrecords_train_set.take(2):
    print("X =", X_batch)
    print("y =", y_batch)
    print()
```

```
X = tf.Tensor(
[[-0.02417601  0.5043504   0.00212128 -0.30513623 -0.24666096 -0.03432783
   1.0838586  -1.2322438 ]
 [-0.02417601  0.5043504   0.00212128 -0.30513623 -0.24666096 -0.03432783
   1.0838586  -1.2322438 ]
 [-0.02417601  0.5043504   0.00212128 -0.30513623 -0.24666096 -0.03432783
   1.0838586  -1.2322438 ]], shape=(3, 8), dtype=float32)
y = tf.Tensor(
[[1.598]
 [1.598]
 [1.598]], shape=(3, 1), dtype=float32)

X = tf.Tensor(
[[-0.02417601  0.5043504   0.00212128 -0.30513623 -0.24666096 -0.03432783
   1.0838586  -1.2322438 ]
 [-0.02417601  0.5043504   0.00212128 -0.30513623 -0.24666096 -0.03432783
   1.0838586  -1.2322438 ]
 [ 0.4422318  -1.7106786   0.7877379   0.03910036 -0.5277444  -0.11205269
  -0.5417641   1.1566417 ]], shape=(3, 8), dtype=float32)
y = tf.Tensor(
[[1.598]
 [1.598]
 [1.745]], shape=(3, 1), dtype=float32)
```

2.4)

Create one dataset for each dataset (`tfrecords_train_set` , `tfrecords_valid_set` and `tfrecords_test_set`), and build, train and evaluate a Keras model using them.

In [58]:

```
batch_size = 32
tfrecords_train_set = tfrecords_reader_dataset("my_train", batch_size)
tfrecords_valid_set = tfrecords_reader_dataset("my_valid", batch_size)
tfrecords_test_set = tfrecords_reader_dataset("my_test", batch_size)
```

In [59]:
```
model = keras.models.Sequential([
    keras.layers.Dense(30, activation="relu", input_shape=[n_inputs]),
    keras.layers.Dense(1),
])
```

In [60]:
```
model.compile(loss="mse", optimizer="sgd")
```

In [61]:
```
model.fit(tfrecords_train_set, steps_per_epoch=len(X_train) // batch_size, epo
chs=10,
          validation_data=tfrecords_valid_set, validation_steps=len(X_valid)
// batch_size)
```

```
Epoch 1/10
362/362==============================] - 1s 4ms/step - loss: 2.4174 - val_los
s: 22.3445
Epoch 2/10
362/362==============================] - 1s 3ms/step - loss: 0.7010 - val_los
s: 32.5775
Epoch 3/10
362/362==============================] - 1s 3ms/step - loss: 0.6279 - val_los
s: 44.5139
Epoch 4/10
362/362==============================] - 1s 3ms/step - loss: 0.5927 - val_los
s: 54.7653
Epoch 5/10
362/362==============================] - 1s 3ms/step - loss: 0.5654 - val_los
s: 64.3574
Epoch 6/10
362/362==============================] - 1s 3ms/step - loss: 0.5405 - val_los
s: 72.7909
Epoch 7/10
362/362==============================] - 1s 3ms/step - loss: 0.5178 - val_los
s: 81.0842
Epoch 8/10
362/362==============================] - 1s 3ms/step - loss: 0.4959 - val_los
s: 88.7506
Epoch 9/10
362/362==============================] - 1s 3ms/step - loss: 0.4806 - val_los
s: 95.3530
Epoch 10/10
362/362==============================] - 1s 3ms/step - loss: 0.4697 - val_los
s: 100.9304
```

Out[61]:
```
<tensorflow.python.keras.callbacks.History at 0x7fb858039d68>
```

In [62]:
```
model.evaluate(tfrecords_test_set, steps=len(X_test) // batch_size)
```

```
161/161==============================] - 0s 2ms/step - loss: 0.8726
```

Out[62]:
```
0.8726394042742919
```

In [63]:

```
new_set = test_set.map(lambda X, y: X)
model.predict(new_set, steps=len(X_test) // batch_size)
```

Out[63]:

```
array([[2.2543297],
       [1.1187534],
       [1.782342 ],
       ...,
       [2.366443 ],
       [1.3198417],
       [2.3221636]], dtype=float32)
```