```
{
"cells": [
 {
  "cell_type": "markdown",
  "metadata": {},
  "source": [
   "# Loading and Preprocessing Data"
  ]
 },
 {
  "cell_type": "markdown",
  "metadata": {},
  "source": [
   "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.\n",
   "\n",
   "<table align=\"left\">\n",
   "  <td>\n",
   "    <a target=\"_blank\" href=\"https://colab.research.google.com/github/ageron/tf2_course/blob/master/03_loading_and_preprocessing_data.ipynb\"><img src=\"https://www.tensorflow.org/images/colab_logo_32px.png\" />Run in Google Colab</a>\n",
   "  </td>\n",
   "</table>"
  ]
 },
 {
  "cell_type": "markdown",
  "metadata": {},
  "source": [
```

```
  "## Imports"
 ]
},
{
 "cell_type": "code",
 "execution_count": null,
 "metadata": {},
 "outputs": [],
 "source": [
  "%matplotlib inline"
 ]
},
{
 "cell_type": "code",
 "execution_count": null,
 "metadata": {},
 "outputs": [],
 "source": [
  "import matplotlib as mpl\n",
  "import matplotlib.pyplot as plt\n",
  "import numpy as np\n",
  "import os\n",
  "import pandas as pd\n",
  "import sklearn\n",
  "import sys\n",
  "import tensorflow as tf\n",
  "from tensorflow import keras\n",
  "import time"
 ]
```

```
},
{
 "cell_type": "code",
 "execution_count": null,
 "metadata": {},
 "outputs": [],
 "source": [
  "print(\"python\", sys.version)\n",
  "for module in mpl, np, pd, sklearn, tf, keras:\n",
  "    print(module._name, module.__version_)"
 ]
},
{
 "cell_type": "code",
 "execution_count": null,
 "metadata": {},
 "outputs": [],
 "source": [
  "assert sys.version_info >= (3, 5) # Python ≥3.5 required\n",
  "assert tf._version_ >= \"2.0\"    # TensorFlow ≥2.0 required"
 ]
},
{
 "cell_type": "markdown",
 "metadata": {},
 "source": [
  "## Code examples"
 ]
},
```

```
{
 "cell_type": "markdown",
 "metadata": {},
 "source": [
  "You can browse through the code examples or jump directly to the exercises."
 ]
},
{
 "cell_type": "code",
 "execution_count": null,
 "metadata": {},
 "outputs": [],
 "source": [
  "dataset = tf.data.Dataset.from_tensor_slices(np.arange(10))\n",
  "dataset"
 ]
},
{
 "cell_type": "code",
 "execution_count": null,
 "metadata": {},
 "outputs": [],
 "source": [
  "for item in dataset:\n",
  "    print(item)"
 ]
},
{
 "cell_type": "code",
```

```
 "execution_count": null,
 "metadata": {},
 "outputs": [],
 "source": [
  "dataset = dataset.repeat(3).batch(7)"
 ]
},
{
 "cell_type": "code",
 "execution_count": null,
 "metadata": {
  "tags": [
   "raises-exception"
  ]
 },
 "outputs": [],
 "source": [
  "for item in dataset:\n",
  "    print(item)"
 ]
},
{
 "cell_type": "code",
 "execution_count": null,
 "metadata": {},
 "outputs": [],
 "source": [
  "dataset = dataset.interleave(\n",
  "    lambda v: tf.data.Dataset.from_tensor_slices(v),\n",
```

```
  "    cycle_length=3,\n",
  "    block_length=2)"
 ]
},
{
 "cell_type": "code",
 "execution_count": null,
 "metadata": {
  "tags": [
   "raises-exception"
  ]
 },
 "outputs": [],
 "source": [
  "for item in dataset:\n",
  "    print(item.numpy(), end=\" \")"
 ]
},
{
 "cell_type": "code",
 "execution_count": null,
 "metadata": {},
 "outputs": [],
 "source": [
  "X = np.array([[2, 3], [4, 5], [6, 7]])\n",
  "y = np.array([\"cat\", \"dog\", \"fox\"])\n",
  "dataset = tf.data.Dataset.from_tensor_slices((X, y))\n",
  "dataset"
 ]
```

```
},
{
 "cell_type": "code",
 "execution_count": null,
 "metadata": {},
 "outputs": [],
 "source": [
  "for item_x, item_y in dataset:\n",
  "    print(item_x.numpy(), item_y.numpy())"
 ]
},
{
 "cell_type": "code",
 "execution_count": null,
 "metadata": {},
 "outputs": [],
 "source": [
  "dataset = tf.data.Dataset.from_tensor_slices({\"features\": X, \"label\": y})\n",
  "dataset"
 ]
},
{
 "cell_type": "code",
 "execution_count": null,
 "metadata": {},
 "outputs": [],
 "source": [
  "for item in dataset:\n",
  "    print(item[\"features\"].numpy(), item[\"label\"].numpy())"
```

```
 ]
},
{
 "cell_type": "markdown",
 "metadata": {},
 "source": [
  "## Split the California dataset to multiple CSV files"
 ]
},
{
 "cell_type": "markdown",
 "metadata": {},
 "source": [
   "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:"
 ]
},
{
 "cell_type": "code",
 "execution_count": null,
 "metadata": {},
 "outputs": [],
 "source": [
  "from sklearn.datasets import fetch_california_housing\n",
  "from sklearn.model_selection import train_test_split\n",
  "from sklearn.preprocessing import StandardScaler\n",
  "\n",
  "housing = fetch_california_housing()\n",
  "X_train_full, X_test, y_train_full, y_test = train_test_split(\n",
```

```
   "    housing.data, housing.target.reshape(-1, 1), random_state=42)\n",
   "X_train, X_valid, y_train, y_valid = train_test_split(\n",
   "    X_train_full, y_train_full, random_state=42)\n",
   "\n",
   "scaler = StandardScaler()\n",
   "X_train_scaled = scaler.fit_transform(X_train)\n",
   "X_valid_scaled = scaler.transform(X_valid)\n",
   "X_test_scaled = scaler.transform(X_test)"
  ]
 },
 {
  "cell_type": "markdown",
  "metadata": {},
  "source": [
   "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:"
  ]
 },
 {
  "cell_type": "code",
  "execution_count": null,
  "metadata": {},
  "outputs": [],
  "source": [
   "def save_to_multiple_csv_files(data, name_prefix, header=None, n_parts=10):\n",
   "    housing_dir = os.path.join(\"datasets\", \"housing\")\n",
   "    os.makedirs(housing_dir, exist_ok=True)\n",
   "    path_format = os.path.join(housing_dir, \"my_{}_{:02d}.csv\")\n",
```

```
   "\n",
   "    filenames = []\n",
   "    m = len(data)\n",
   "    for file_idx, row_indices in enumerate(np.array_split(np.arange(m), n_parts)):\n",
   "        part_csv = path_format.format(name_prefix, file_idx)\n",
   "        filenames.append(part_csv)\n",
   "        with open(part_csv, \"wt\", encoding=\"utf-8\") as f:\n",
   "            if header is not None:\n",
   "                f.write(header)\n",
   "                f.write(\"\\n\")\n",
   "            for row_idx in row_indices:\n",
   "                f.write(\",\".join([repr(col) for col in data[row_idx]]))\n",
   "                f.write(\"\\n\")\n",
   "    return filenames"
  ]
 },
 {
  "cell_type": "code",
  "execution_count": null,
  "metadata": {},
  "outputs": [],
  "source": [
   "train_data = np.c_[X_train_scaled, y_train]\n",
   "valid_data = np.c_[X_valid_scaled, y_valid]\n",
   "test_data = np.c_[X_test_scaled, y_test]\n",
   "header_cols = [\"Scaled\" + name for name in housing.feature_names] +
[\"MedianHouseValue\"]\n",
   "header = \",\".join(header_cols)\n",
   "\n",
```

```
  "train_filenames = save_to_multiple_csv_files(train_data, \"train\", header, n_parts=20)\n",
  "valid_filenames = save_to_multiple_csv_files(valid_data, \"valid\", header, n_parts=10)\n",
  "test_filenames = save_to_multiple_csv_files(test_data, \"test\", header, n_parts=10)"
 ]
},
{
 "cell_type": "markdown",
 "metadata": {},
 "source": [
  "Okay, now let's take a peek at the first few lines of one of these CSV files:"
 ]
},
{
 "cell_type": "code",
 "execution_count": null,
 "metadata": {},
 "outputs": [],
 "source": [
  "with open(train_filenames[0]) as f:\n",
  "    for i in range(3):\n",
  "        print(f.readline(), end=\"\")"
 ]
},
{
 "cell_type": "markdown",
 "metadata": {},
 "source": [
  "![Exercise](https://c1.staticflickr.com/9/8101/8553474140_c50cf08708_b.jpg)"
 ]
```

```
},
{
 "cell_type": "markdown",
 "metadata": {},
 "source": [
  "## Exercise 1 – Data API"
 ]
},
{
 "cell_type": "markdown",
 "metadata": {},
 "source": [
  "### 1.1)\n",
   "Use tf.data.Dataset.list_files() to create a dataset that will simply list the training filenames. Iterate through its items and print them."
 ]
},
{
 "cell_type": "code",
 "execution_count": null,
 "metadata": {},
 "outputs": [],
 "source": []
},
{
 "cell_type": "code",
 "execution_count": null,
 "metadata": {},
 "outputs": [],
```

```
 "source": []
},
{
 "cell_type": "code",
 "execution_count": null,
 "metadata": {},
 "outputs": [],
 "source": []
},
{
 "cell_type": "markdown",
 "metadata": {},
 "source": [
  "### 1.2)\n",
   "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."
 ]
},
{
 "cell_type": "code",
 "execution_count": null,
 "metadata": {},
 "outputs": [],
 "source": []
},
{
```

```
  "cell_type": "code",
  "execution_count": null,
  "metadata": {},
  "outputs": [],
  "source": []
 },
 {
  "cell_type": "code",
  "execution_count": null,
  "metadata": {},
  "outputs": [],
  "source": []
 },
 {
  "cell_type": "markdown",
  "metadata": {},
  "source": [
   "### 1.3)\n",
   "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."
  ]
 },
 {
  "cell_type": "code",
  "execution_count": null,
  "metadata": {},
  "outputs": [],
  "source": []
```

```
 },
 {
  "cell_type": "code",
  "execution_count": null,
  "metadata": {},
  "outputs": [],
  "source": []
 },
 {
  "cell_type": "code",
  "execution_count": null,
  "metadata": {},
  "outputs": [],
  "source": []
 },
 {
  "cell_type": "markdown",
  "metadata": {},
  "source": [
   "### 1.4)\n",
   "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.).\n",
   "* 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)."
  ]
 },
 {
```

```
 "cell_type": "code",
 "execution_count": null,
 "metadata": {},
 "outputs": [],
 "source": [
  "record_defaults=[0, np.nan, tf.constant(np.nan, dtype=tf.float64), \"Hello\", tf.constant([])]\n",
  "parsed_fields = tf.io.decode_csv('1,2,3,4,5', record_defaults)\n",
  "parsed_fields"
 ]
},
{
 "cell_type": "code",
 "execution_count": null,
 "metadata": {},
 "outputs": [],
 "source": []
},
{
 "cell_type": "code",
 "execution_count": null,
 "metadata": {},
 "outputs": [],
 "source": []
},
{
 "cell_type": "code",
 "execution_count": null,
 "metadata": {},
 "outputs": [],
```

```
  "source": []
  },
  {
  "cell_type": "markdown",
  "metadata": {},
  "source": [
   "### 1.5)\n",
   "Now you are ready to create a function to parse a CSV line:\n",
   "* Create a parse_csv_line() function that takes a single line as argument.\n",
   "* Call tf.io.decode_csv() to parse that line.\n",
   "* Call tf.stack() to create a single tensor containing all the input features (i.e., all fields except the last one).\n",
   "* 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].\n",
   "* Return a tuple with both tensors (input features and labels).\n",
   "* Try calling it on a single line from one of the CSV files."
  ]
  },
  {
  "cell_type": "code",
  "execution_count": null,
  "metadata": {},
  "outputs": [],
  "source": []
  },
  {
  "cell_type": "code",
  "execution_count": null,
  "metadata": {},
```

```
  "outputs": [],
  "source": []
 },
 {
  "cell_type": "code",
  "execution_count": null,
  "metadata": {},
  "outputs": [],
  "source": []
 },
 {
  "cell_type": "markdown",
  "metadata": {},
  "source": [
   "### 1.6)\n",
   "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.\n",
   "\n",
   "*Tips*:\n",
   "* 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).\n",
   "* 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.\n",
   "* 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).\n",
   "* 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).\n",
```

"* 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).\n",

"* 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).\n",

"* Return the resulting dataset.\n",

"* Give every argument a reasonable default value (except for the filenames).\n",

"* Test your function by calling it with a small batch size and printing the first couple of batches.\n",

"* 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."

]

},

{

"cell_type": "code",

"execution_count": null,

"metadata": {},

"outputs": [],

"source": []

},

{

"cell_type": "code",

"execution_count": null,

"metadata": {},

"outputs": [],

"source": []

},

{

"cell_type": "code",

"execution_count": null,

```
 "metadata": {},
 "outputs": [],
 "source": []
},
{
 "cell_type": "markdown",
 "metadata": {},
 "source": [
  "### 1.7)\n",
  "Build a training set, a validation set and a test set using your csv_reader_dataset() function."
 ]
},
{
 "cell_type": "code",
 "execution_count": null,
 "metadata": {},
 "outputs": [],
 "source": []
},
{
 "cell_type": "code",
 "execution_count": null,
 "metadata": {},
 "outputs": [],
 "source": []
},
{
 "cell_type": "code",
 "execution_count": null,
```

```
  "metadata": {},
  "outputs": [],
  "source": []
 },
 {
  "cell_type": "markdown",
  "metadata": {},
  "source": [
   "### 1.8)\n",
   "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.\n",
   "\n",
   "*Tips*\n",
   "* 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.\n",
   "* Similarly, pass the validation dataset instead of (X_valid_scaled, y_valid) and y_valid, and set the validation_steps.\n",
   "* For the evaluate() and predict() methods, you need to pass the test dataset, and specify the steps argument.\n",
   "* 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))."
  ]
 },
 {
  "cell_type": "code",
  "execution_count": null,
  "metadata": {},
  "outputs": [],
  "source": []
```

```
 },
 {
  "cell_type": "code",
  "execution_count": null,
  "metadata": {},
  "outputs": [],
  "source": []
 },
 {
  "cell_type": "code",
  "execution_count": null,
  "metadata": {},
  "outputs": [],
  "source": []
 },
 {
  "cell_type": "markdown",
  "metadata": {},
  "source": [
   "![Exercise
solution](https://camo.githubusercontent.com/250388fde3fac9135ead9471733ee28e049f7a37/687474
70733a2f2f75706c6f61642e77696b696d656469612e6f72672f77696b6970656469612f636f6d6d6f6e732
f302f30362f46696c6f735f736567756e646f5f6c6f676f5f253238666c69707065642532392e6a7067)"
  ]
 },
 {
  "cell_type": "markdown",
  "metadata": {},
  "source": [
   "## Exercise 1 – Solution"
```

```
 ]
},
{
 "cell_type": "markdown",
 "metadata": {},
 "source": [
  "### 1.1)\n",
   "Use tf.data.Dataset.list_files() to create a dataset that will simply list the training filenames. Iterate through its items and print them."
 ]
},
{
 "cell_type": "code",
 "execution_count": null,
 "metadata": {},
 "outputs": [],
 "source": [
  "filename_dataset = tf.data.Dataset.list_files(train_filenames)"
 ]
},
{
 "cell_type": "code",
 "execution_count": null,
 "metadata": {},
 "outputs": [],
 "source": [
  "for filename in filename_dataset:\n",
  "    print(filename)"
 ]
```

},

{

"cell_type": "markdown",

"metadata": {},

"source": [

"### 1.2)\n",

"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."

]

},

{

"cell_type": "code",

"execution_count": null,

"metadata": {},

"outputs": [],

"source": [

"n_readers = 5\n",

"dataset = filename_dataset.interleave(\n",

"   lambda filename: tf.data.TextLineDataset(filename),\n",

"   cycle_length=n_readers)"

]

},

{

"cell_type": "code",

"execution_count": null,

"metadata": {},

```
 "outputs": [],
 "source": [
  "for line in dataset.take(15):\n",
  "    print(line.numpy())"
 ]
},
{
 "cell_type": "markdown",
 "metadata": {},
 "source": [
  "### 1.3)\n",
  "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."
 ]
},
{
 "cell_type": "code",
 "execution_count": null,
 "metadata": {},
 "outputs": [],
 "source": [
  "dataset = filename_dataset.interleave(\n",
  "    lambda filename: tf.data.TextLineDataset(filename).skip(1),\n",
  "    cycle_length=n_readers)"
 ]
},
{
 "cell_type": "code",
```

```
  "execution_count": null,
  "metadata": {},
  "outputs": [],
  "source": [
   "for line in dataset.take(5):\n",
   "    print(line.numpy())"
  ]
 },
 {
  "cell_type": "markdown",
  "metadata": {},
  "source": [
   "### 1.4)\n",
   "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 removing some field values, etc.)."
  ]
 },
 {
  "cell_type": "markdown",
  "metadata": {},
  "source": [
   "Notice that field 4 is interpreted as a string."
  ]
 },
 {
  "cell_type": "code",
  "execution_count": null,
  "metadata": {},
  "outputs": [],
```

```
 "source": [
  "record_defaults=[0, np.nan, tf.constant(np.nan, dtype=tf.float64), \"Hello\", tf.constant([])]\n",
  "parsed_fields = tf.io.decode_csv('1,2,3,4,5', record_defaults)\n",
  "parsed_fields"
 ]
},
{
 "cell_type": "markdown",
 "metadata": {},
 "source": [
  "Notice that all missing fields are replaced with their default value, when provided:"
 ]
},
{
 "cell_type": "code",
 "execution_count": null,
 "metadata": {},
 "outputs": [],
 "source": [
  "parsed_fields = tf.io.decode_csv(',,,,5', record_defaults)\n",
  "parsed_fields"
 ]
},
{
 "cell_type": "markdown",
 "metadata": {},
 "source": [
  "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:"
```

```
 ]
},
{
 "cell_type": "code",
 "execution_count": null,
 "metadata": {},
 "outputs": [],
 "source": [
  "try:\n",
  "    parsed_fields = tf.io.decode_csv(',,,,', record_defaults)\n",
  "except tf.errors.InvalidArgumentError as ex:\n",
  "    print(ex)"
 ]
},
{
 "cell_type": "markdown",
 "metadata": {},
 "source": [
  "The number of fields should match exactly the number of fields in the record_defaults:"
 ]
},
{
 "cell_type": "code",
 "execution_count": null,
 "metadata": {},
 "outputs": [],
 "source": [
  "try:\n",
  "    parsed_fields = tf.io.decode_csv('1,2,3,4,5,6,7', record_defaults)\n",
```

```
   "except tf.errors.InvalidArgumentError as ex:\n",
   "    print(ex)"
  ]
 },
 {
  "cell_type": "markdown",
  "metadata": {},
  "source": [
   "### 1.5)\n",
   "Now you are ready to create a function to parse a CSV line:\n",
   "* Create a parse_csv_line() function that takes a single line as argument.\n",
   "* Call tf.io.decode_csv() to parse that line.\n",
   "* Call tf.stack() to create a single tensor containing all the input features (i.e., all fields except the last one).\n",
   "* 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].\n",
   "* Return a tuple with both tensors (input features and labels).\n",
   "* Try calling it on a single line from one of the CSV files."
  ]
 },
 {
  "cell_type": "code",
  "execution_count": null,
  "metadata": {
   "scrolled": false
  },
  "outputs": [],
  "source": [
   "n_inputs = X_train.shape[1]\n",
```

```
   "\n",
   "def parse_csv_line(line, n_inputs=n_inputs):\n",
   "    defs = [tf.constant(np.nan)] * (n_inputs + 1)\n",
   "    fields = tf.io.decode_csv(line, record_defaults=defs)\n",
   "    x = tf.stack(fields[:-1])\n",
   "    y = tf.stack(fields[-1:])\n",
   "    return x, y"
  ]
 },
 {
  "cell_type": "code",
  "execution_count": null,
  "metadata": {},
  "outputs": [],
  "source": [
   "parse_csv_line(b'-0.739840972632228,-0.3658395634576743,-0.784679995482575,0.07414513752253027,0.7544706668961565,0.407700592469922,-0.686992593958441,0.6019005115704453,2.0')"
  ]
 },
 {
  "cell_type": "markdown",
  "metadata": {},
  "source": [
   "### 1.6)\n",
   "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."
  ]
 },
```

```
{
 "cell_type": "code",
 "execution_count": null,
 "metadata": {},
 "outputs": [],
 "source": [
  "def csv_reader_dataset(filenames, n_parse_threads=5, batch_size=32,\n",
  "                       shuffle_buffer_size=10000, n_readers=5):\n",
  "    dataset = tf.data.Dataset.list_files(filenames)\n",
  "    dataset = dataset.repeat()\n",
  "    dataset = dataset.interleave(\n",
  "        lambda filename: tf.data.TextLineDataset(filename).skip(1),\n",
  "        cycle_length=n_readers)\n",
  "    dataset = dataset.shuffle(shuffle_buffer_size)\n",
  "    dataset = dataset.map(parse_csv_line, num_parallel_calls=n_parse_threads)\n",
  "    dataset = dataset.batch(batch_size)\n",
  "    return dataset.prefetch(1)"
 ]
},
{
 "cell_type": "markdown",
 "metadata": {},
 "source": [
  "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):"
 ]
},
{
 "cell_type": "code",
```

```
"execution_count": null,
"metadata": {},
"outputs": [],
"source": [
 "def csv_reader_dataset(filenames, batch_size=32,\n",
 "                       shuffle_buffer_size=10000, n_readers=5):\n",
 "    dataset = tf.data.Dataset.list_files(filenames)\n",
 "    dataset = dataset.repeat()\n",
 "    dataset = dataset.interleave(\n",
 "        lambda filename: tf.data.TextLineDataset(filename).skip(1),\n",
 "        cycle_length=n_readers)\n",
 "    dataset = dataset.shuffle(shuffle_buffer_size)\n",
 "    dataset = dataset.apply(\n",
 "        tf.data.experimental.map_and_batch(\n",
 "            parse_csv_line,\n",
 "            batch_size,\n",
 "            num_parallel_calls=tf.data.experimental.AUTOTUNE))\n",
 "    return dataset.prefetch(1)"
]
},
{
 "cell_type": "code",
 "execution_count": null,
 "metadata": {},
 "outputs": [],
 "source": [
  "train_set = csv_reader_dataset(train_filenames, batch_size=3)\n",
  "for X_batch, y_batch in train_set.take(2):\n",
  "    print(\"X =\", X_batch)\n",
```

```
  "    print(\"y =\", y_batch)\n",
  "    print()"
 ]
},
{
 "cell_type": "markdown",
 "metadata": {},
 "source": [
  "### 1.7)\n",
  "Build a training set, a validation set and a test set using your csv_reader_dataset() function."
 ]
},
{
 "cell_type": "code",
 "execution_count": null,
 "metadata": {},
 "outputs": [],
 "source": [
  "batch_size = 32\n",
  "train_set = csv_reader_dataset(train_filenames, batch_size)\n",
  "valid_set = csv_reader_dataset(valid_filenames, batch_size)\n",
  "test_set = csv_reader_dataset(test_filenames, batch_size)"
 ]
},
{
 "cell_type": "markdown",
 "metadata": {},
 "source": [
  "### 1.8)\n",
```

"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."

]

},

{

"cell_type": "code",

"execution_count": null,

"metadata": {},

"outputs": [],

"source": [

"model = keras.models.Sequential([\n",

"   keras.layers.Dense(30, activation=\"relu\", input_shape=[n_inputs]),\n",

"   keras.layers.Dense(1),\n",

"])"

]

},

{

"cell_type": "code",

"execution_count": null,

"metadata": {},

"outputs": [],

"source": [

"model.compile(loss=\"mse\", optimizer=keras.optimizers.SGD(lr=1e-3))"

]

},

{

"cell_type": "code",

"execution_count": null,

"metadata": {},

```
 "outputs": [],
 "source": [
  "model.fit(train_set, steps_per_epoch=len(X_train) // batch_size, epochs=10,\n",
  "          validation_data=valid_set, validation_steps=len(X_valid) // batch_size)"
 ]
},
{
 "cell_type": "code",
 "execution_count": null,
 "metadata": {},
 "outputs": [],
 "source": [
  "model.evaluate(test_set, steps=len(X_test) // batch_size)"
 ]
},
{
 "cell_type": "code",
 "execution_count": null,
 "metadata": {
  "scrolled": true
 },
 "outputs": [],
 "source": [
  "new_set = test_set.map(lambda X, y: X)\n",
  "model.predict(new_set, steps=len(X_test) // batch_size)"
 ]
},
{
 "cell_type": "markdown",
```

```
 "metadata": {},
 "source": [
  "![Exercise](https://c1.staticflickr.com/9/8101/8553474140_c50cf08708_b.jpg)"
 ]
},
{
 "cell_type": "markdown",
 "metadata": {},
 "source": [
  "## Exercise 2 – The TFRecord binary format"
 ]
},
{
 "cell_type": "markdown",
 "metadata": {},
 "source": [
  "### Code examples"
 ]
},
{
 "cell_type": "markdown",
 "metadata": {},
 "source": [
  "You can walk through these code examples or jump down to the [actual exercise](#Actual-exercise) below."
 ]
},
{
 "cell_type": "code",
```

```
 "execution_count": null,
 "metadata": {},
 "outputs": [],
 "source": [
  "favorite_books = [name.encode(\"utf-8\")\n",
  "                  for name in [\"Arluk\", \"Fahrenheit 451\", \"L'étranger\"]]\n",
  "favorite_books = tf.train.BytesList(value=favorite_books)\n",
  "favorite_books"
 ]
},
{
 "cell_type": "code",
 "execution_count": null,
 "metadata": {},
 "outputs": [],
 "source": [
  "hours_per_month = tf.train.FloatList(value=[15.5, 9.5, np.nan, 6.0, 9.0])\n",
  "hours_per_month"
 ]
},
{
 "cell_type": "code",
 "execution_count": null,
 "metadata": {},
 "outputs": [],
 "source": [
  "age = tf.train.Int64List(value=[42])\n",
  "age"
 ]
```

```
},
{
 "cell_type": "code",
 "execution_count": null,
 "metadata": {},
 "outputs": [],
 "source": [
  "coordinates = tf.train.FloatList(value=[1.2834, 103.8607])\n",
  "coordinates"
 ]
},
{
 "cell_type": "code",
 "execution_count": null,
 "metadata": {},
 "outputs": [],
 "source": [
  "features = tf.train.Features(\n",
  "    feature={\n",
  "        \"favorite_books\": tf.train.Feature(bytes_list=favorite_books),\n",
  "        \"hours_per_month\": tf.train.Feature(float_list=hours_per_month),\n",
  "        \"age\": tf.train.Feature(int64_list=age),\n",
  "        \"coordinates\": tf.train.Feature(float_list=coordinates),\n",
  "    }\n",
  ")\n",
  "features"
 ]
},
{
```

```
 "cell_type": "code",
 "execution_count": null,
 "metadata": {},
 "outputs": [],
 "source": [
  "example = tf.train.Example(features=features)\n",
  "example"
 ]
},
{
 "cell_type": "code",
 "execution_count": null,
 "metadata": {},
 "outputs": [],
 "source": [
  "serialized_example = example.SerializeToString()\n",
  "serialized_example"
 ]
},
{
 "cell_type": "code",
 "execution_count": null,
 "metadata": {},
 "outputs": [],
 "source": [
  "filename = \"my_reading_data.tfrecords\"\n",
  "with tf.io.TFRecordWriter(filename) as writer:\n",
  "    for i in range(3): # you should save different examples instead! :)\n",
  "        writer.write(serialized_example)"
```

```
 ]
},
{
 "cell_type": "code",
 "execution_count": null,
 "metadata": {},
 "outputs": [],
 "source": [
  "for serialized_example_tensor in tf.data.TFRecordDataset([filename]):\n",
  "    print(serialized_example_tensor)"
 ]
},
{
 "cell_type": "code",
 "execution_count": null,
 "metadata": {},
 "outputs": [],
 "source": [
  "filename = \"my_reading_data.tfrecords\"\n",
  "options = tf.io.TFRecordOptions(compression_type=\"GZIP\")\n",
  "with tf.io.TFRecordWriter(filename, options) as writer:\n",
  "    for i in range(3): # you should save different examples instead! :)\n",
  "        writer.write(serialized_example)"
 ]
},
{
 "cell_type": "code",
 "execution_count": null,
 "metadata": {},
```

```
 "outputs": [],
 "source": [
  "dataset = tf.data.TFRecordDataset([filename], compression_type=\"GZIP\")\n",
  "for serialized_example_tensor in dataset:\n",
  "    print(serialized_example_tensor)"
 ]
},
{
 "cell_type": "code",
 "execution_count": null,
 "metadata": {},
 "outputs": [],
 "source": [
  "expected_features = {\n",
  "    \"favorite_books\": tf.io.VarLenFeature(dtype=tf.string),\n",
  "    \"hours_per_month\": tf.io.VarLenFeature(dtype=tf.float32),\n",
  "    \"age\": tf.io.FixedLenFeature([], dtype=tf.int64),\n",
  "    \"coordinates\": tf.io.FixedLenFeature([2], dtype=tf.float32),\n",
  "}\n",
  "\n",
  "for serialized_example_tensor in tf.data.TFRecordDataset(\n",
  "        [filename], compression_type=\"GZIP\"):\n",
  "    example = tf.io.parse_single_example(serialized_example_tensor,\n",
  "                                         expected_features)\n",
  "    books = tf.sparse.to_dense(example[\"favorite_books\"],\n",
  "                               default_value=b\"\")\n",
  "    for book in books:\n",
  "        print(book.numpy().decode('UTF-8'), end=\"\\t\")\n",
  "    print()"
```

```
 ]
},
{
 "cell_type": "markdown",
 "metadata": {},
 "source": [
  "## Actual exercise"
 ]
},
{
 "cell_type": "markdown",
 "metadata": {},
 "source": [
  "### 2.1)\n",
  "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.\n",
  "\n",
  "*Tips*:\n",
  "* 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.\n",
  "* to format 19 as \"00019\", you can use \"{:05d}\".format(19)."
 ]
},
{
 "cell_type": "code",
 "execution_count": null,
 "metadata": {},
```

```
 "outputs": [],
 "source": []
},
{
 "cell_type": "code",
 "execution_count": null,
 "metadata": {},
 "outputs": [],
 "source": []
},
{
 "cell_type": "code",
 "execution_count": null,
 "metadata": {},
 "outputs": [],
 "source": []
},
{
 "cell_type": "markdown",
 "metadata": {},
 "source": [
  "### 2.2)\n",
  "Use this function to write the training set, validation set and test set to multiple TFRecord files."
 ]
},
{
 "cell_type": "code",
 "execution_count": null,
 "metadata": {},
```

```
  "outputs": [],
  "source": []
 },
 {
  "cell_type": "code",
  "execution_count": null,
  "metadata": {},
  "outputs": [],
  "source": []
 },
 {
  "cell_type": "code",
  "execution_count": null,
  "metadata": {},
  "outputs": [],
  "source": []
 },
 {
  "cell_type": "markdown",
  "metadata": {},
  "source": [
   "### 2.3)\n",
   "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.\n",
   "\n",
   "*Tips*:\n",
   "* 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()).\n",
   "* The parsing function should return (input features, label), not a tf.train.Example."
```

```
 ]
},
{
 "cell_type": "code",
 "execution_count": null,
 "metadata": {},
 "outputs": [],
 "source": []
},
{
 "cell_type": "code",
 "execution_count": null,
 "metadata": {},
 "outputs": [],
 "source": []
},
{
 "cell_type": "code",
 "execution_count": null,
 "metadata": {},
 "outputs": [],
 "source": []
},
{
 "cell_type": "code",
 "execution_count": null,
 "metadata": {},
 "outputs": [],
 "source": []
},
{
 "cell_type": "markdown",
 "metadata": {},
 "source": [
  "### 2.4)\n",
```

"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."

]

},

{

"cell_type": "code",

"execution_count": null,

"metadata": {},

"outputs": [],

"source": []

},

{

"cell_type": "code",

"execution_count": null,

"metadata": {},

"outputs": [],

"source": []

},

{

"cell_type": "code",

"execution_count": null,

"metadata": {},

"outputs": [],

"source": []

},

{

"cell_type": "markdown",

"metadata": {},

"source": [

"![Exercise solution](https://camo.githubusercontent.com/250388fde3fac9135ead9471733ee28e049f7a37/68747470733a2f2f75706c6f61642e77696b696d656469612e6f72672f77696b6970656469612f636f6d6d6f6e732f302f30362f46696c6f735f736567756e646f5f6c6f676f5f253238666c69707065642532392e6a7067)"

]

},

{

"cell_type": "markdown",

"metadata": {},

"source": [

"## Exercise 2 – Solution"

]

},

{

"cell_type": "markdown",

"metadata": {},

"source": [

"### 2.1)\n",

"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."

]

},

{

"cell_type": "code",

"execution_count": null,

"metadata": {},

"outputs": [],

"source": [

```
 "def serialize_example(x, y):\n",
 "    input_features = tf.train.FloatList(value=x)\n",
 "    label = tf.train.FloatList(value=y)\n",
 "    features = tf.train.Features(\n",
 "        feature = {\n",
 "            \"input_features\": tf.train.Feature(float_list=input_features),\n",
 "            \"label\": tf.train.Feature(float_list=label),\n",
 "        }\n",
 "    )\n",
 "    example = tf.train.Example(features=features)\n",
 "    return example.SerializeToString()"
 ]
},
{
 "cell_type": "code",
 "execution_count": null,
 "metadata": {},
 "outputs": [],
 "source": [
 "def csv_to_tfrecords(filename, csv_reader_dataset, n_shards, steps_per_shard,\n",
 "                     compression_type=None):\n",
 "    options = tf.io.TFRecordOptions(compression_type=compression_type)\n",
 "    for shard in range(n_shards):\n",
 "        path = \"{}_{:05d}-of-{:05d}.tfrecords\".format(filename, shard, n_shards)\n",
 "        with tf.io.TFRecordWriter(path, options) as writer:\n",
 "            for X_batch, y_batch in csv_reader_dataset.take(steps_per_shard):\n",
 "                for x_instance, y_instance in zip(X_batch, y_batch):\n",
 "                    writer.write(serialize_example(x_instance, y_instance))"
 ]
```

```
},
{
 "cell_type": "markdown",
 "metadata": {},
 "source": [
  "### 2.2)\n",
  "Use this function to write the training set, validation set and test set to multiple TFRecord files."
 ]
},
{
 "cell_type": "code",
 "execution_count": null,
 "metadata": {},
 "outputs": [],
 "source": [
  "batch_size = 32\n",
  "n_shards = 20\n",
  "steps_per_shard = len(X_train) // batch_size // n_shards\n",
  "csv_to_tfrecords(\"my_train.tfrecords\", train_set, n_shards, steps_per_shard)\n",
  "\n",
  "n_shards = 1\n",
  "steps_per_shard = len(X_valid) // batch_size // n_shards\n",
  "csv_to_tfrecords(\"my_valid.tfrecords\", valid_set, n_shards, steps_per_shard)\n",
  "\n",
  "n_shards = 1\n",
  "steps_per_shard = len(X_test) // batch_size // n_shards\n",
  "csv_to_tfrecords(\"my_test.tfrecords\", test_set, n_shards, steps_per_shard)"
 ]
},
```

```
 {
  "cell_type": "markdown",
  "metadata": {},
  "source": [
   "### 2.3)\n",
   "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."
  ]
 },
 {
  "cell_type": "code",
  "execution_count": null,
  "metadata": {},
  "outputs": [],
  "source": [
   "expected_features = {\n",
   "    \"input_features\": tf.io.FixedLenFeature([n_inputs], dtype=tf.float32),\n",
   "    \"label\": tf.io.FixedLenFeature([1], dtype=tf.float32),\n",
   "}\n",
   "\n",
   "def parse_tfrecord(serialized_example):\n",
   "    example = tf.io.parse_single_example(serialized_example,\n",
   "                                         expected_features)\n",
   "    return example[\"input_features\"], example[\"label\"]"
  ]
 },
 {
  "cell_type": "code",
```

```
 "execution_count": null,
 "metadata": {},
 "outputs": [],
 "source": [
  "def tfrecords_reader_dataset(filename, batch_size=32,\n",
  "                             shuffle_buffer_size=10000, n_readers=5):\n",
  "    filenames = [name for name in os.listdir() if name.startswith(filename)\n",
  "                                       and name.endswith(\".tfrecords\")]\n",
  "    dataset = tf.data.Dataset.list_files(filenames)\n",
  "    dataset = dataset.repeat()\n",
  "    dataset = dataset.interleave(\n",
  "        lambda filename: tf.data.TFRecordDataset(filename),\n",
  "        cycle_length=n_readers)\n",
  "    dataset.shuffle(shuffle_buffer_size)\n",
  "    dataset = dataset.apply(\n",
  "        tf.data.experimental.map_and_batch(\n",
  "            parse_tfrecord,\n",
  "            batch_size,\n",
  "            num_parallel_calls=tf.data.experimental.AUTOTUNE))\n",
  "    return dataset.prefetch(1)"
 ]
},
{
 "cell_type": "code",
 "execution_count": null,
 "metadata": {},
 "outputs": [],
 "source": [
  "tfrecords_train_set = tfrecords_reader_dataset(\"my_train\", batch_size=3)\n",
```

```
   "for X_batch, y_batch in tfrecords_train_set.take(2):\n",
   "    print(\"X =\", X_batch)\n",
   "    print(\"y =\", y_batch)\n",
   "    print()"
  ]
 },
 {
  "cell_type": "markdown",
  "metadata": {},
  "source": [
   "### 2.4)\n",
   "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."
  ]
 },
 {
  "cell_type": "code",
  "execution_count": null,
  "metadata": {},
  "outputs": [],
  "source": [
   "batch_size = 32\n",
   "tfrecords_train_set = tfrecords_reader_dataset(\"my_train\", batch_size)\n",
   "tfrecords_valid_set = tfrecords_reader_dataset(\"my_valid\", batch_size)\n",
   "tfrecords_test_set = tfrecords_reader_dataset(\"my_test\", batch_size)"
  ]
 },
 {
  "cell_type": "code",
```

```
 "execution_count": null,
 "metadata": {},
 "outputs": [],
 "source": [
  "model = keras.models.Sequential([\n",
  "    keras.layers.Dense(30, activation=\"relu\", input_shape=[n_inputs]),\n",
  "    keras.layers.Dense(1),\n",
  "])"
 ]
},
{
 "cell_type": "code",
 "execution_count": null,
 "metadata": {},
 "outputs": [],
 "source": [
  "model.compile(loss=\"mse\", optimizer=keras.optimizers.SGD(lr=1e-3))"
 ]
},
{
 "cell_type": "code",
 "execution_count": null,
 "metadata": {},
 "outputs": [],
 "source": [
  "model.fit(tfrecords_train_set, steps_per_epoch=len(X_train) // batch_size, epochs=10,\n",
  "          validation_data=tfrecords_valid_set, validation_steps=len(X_valid) // batch_size)"
 ]
},
```

```
  {
   "cell_type": "code",
   "execution_count": null,
   "metadata": {},
   "outputs": [],
   "source": [
    "model.evaluate(tfrecords_test_set, steps=len(X_test) // batch_size)"
   ]
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "metadata": {
    "scrolled": true
   },
   "outputs": [],
   "source": [
    "new_set = test_set.map(lambda X, y: X)\n",
    "model.predict(new_set, steps=len(X_test) // batch_size)"
   ]
  }
 ],
 "metadata": {
  "kernelspec": {
   "display_name": "Python 3",
   "language": "python",
   "name": "python3"
  },
  "language_info": {
```

```
   "codemirror_mode": {
    "name": "ipython",
    "version": 3
   },
   "file_extension": ".py",
   "mimetype": "text/x-python",
   "name": "python",
   "nbconvert_exporter": "python",
   "pygments_lexer": "ipython3",
   "version": "3.7.10"
  }
},
"nbformat": 4,
"nbformat_minor": 2
}
```