4

Working with Other Data

In the previous chapter, we showed you how to work with TFDS. But, what if you have another type of dataset? In this chapter, we show you how to work with other types of data with TensorFlow. Notebooks for this chapter are located at the following URL: <https://github.com/paperd/tensorflow>.

We need to set up the GPU for each notebook. So, click the Runtime tab, click Change runtime type from the drop-down menu, choose GPU from the Hardware Accelerator drop-down menu, and click Save.

Display the current version of TensorFlow and enable the GPU in Google Colab:

import tensorflow as tf

# display tf version and test if GPU is active

tf.\_\_version\_\_, tf.test.gpu\_device\_name()

The GPU is active if you see '/device:GPU:0' displayed.

# Basic Mechanics

To create an input pipeline, we start with a data source. To construct a dataset from data in memory that TensorFlow can work with, we can use either tf.data.Dataset.from\_tensor\_slices() or tf.data.Dataset.from\_tensors(). The from\_tensor\_slices method creates a dataset with a separate element for each piece of the input tensor. The from\_tensors method combines the input and returns a dataset with a single element. We work exclusively with the from\_tensor\_slices method because it enables us to work with each data element.

Let’s create a simple 1D tensor and make it consumable for TensorFlow:

# create a 1D tensor

ds = tf.data.Dataset.from\_tensor\_slices([8, 3, 0, 8, 2, 1])

ds.element\_spec

We create a TensorSpec from the six element list and make it TensorFlow consumable with the from\_tensor\_slices method. The shape of the dataset is () because we just created a TensorFlow scalar.

Listing 4-1 demonstrates how to iterate the dataset:

# iterate and display tensor values

for elem in ds:

  print(elem.numpy())

print ()

# iterate without numpy method

for elem in ds:

  print(elem)

Listing 4-1. Iterate and display the from\_tensor\_slices dataset

We have a dataset with six tensors. The first loop displays each element in the tensor as numpy values. The second loop displays the raw tensors.

We can also create a Python iterator using iter and consume its elements with the next method:

it = iter(ds)

# display the first element

next(it).numpy()

We see the first element from the tensor displayed.

To view the remaining elements, just keep running the following:

next(it).numpy()

Now, create a tensor with tf.data.Dataset\_from\_tensors():

# create a 1D tensor

ds = tf.data.Dataset.from\_tensors([8, 3, 0, 8, 2, 1])

ds.element\_spec

Notice that the shape is (6,), which means that the single tensor is composed of six elements.

Let’s iterate the single tensor as shown in Listing 4-2:

# iterate and display tensor values

for elem in ds:

  print(elem.numpy())

print ()

# iterate without numpy method

for elem in ds:

  print(elem)

Listing 4-2. Iterate and display the from\_tensors dataset

We have a dataset with a single tensor that contains six elements.

# TensorFlow Dataset Structure

A dataset contains elements that each have the same nested structure and the individual components of the structure can have any type representable by tf.TypeSpec, including tf.Tensor, tf.sparse.SparseTensor, tf.RaggedTensor, tf.TensorArray or tf.data.Dataset. The Dataset.element\_spec property allows us to inspect the type of each element component. The property returns a nested structure of tf.TypeSpec objects that match the structure of the element. The element may be a single component, a tuple of components, or a nested tuple of components.

We can better understand the structure with an example as shown in Listing 4-3:

br = '\n'  # enter a line break in Colab

# create random uniform numbers

scope = tf.random.uniform([4, 10])

print ('shape:', scope.shape, br)

ds = tf.data.Dataset.from\_tensor\_slices(scope)

print (ds.element\_spec, br)

# Let's look at the first element:

it = iter(ds)

# print first element

print ('first element with an iterator:', br)

print (next(it).numpy(), br)

print ('all four elements:', br)

for i, row in enumerate(ds):

  print ('element ' + str(i+1))  # add 1 as index starts at 0

  print (row.numpy(), br)

Listing 4-3. Data structure example

The shape of scope is (4, 10), which means we have a tensor with four elements each of which contain ten randomly generated uniformly distributed numbers between 0 and 1. We generate a TensorFlow consumable dataset from scope with the from\_tensor\_slices method and display the TensorSpec with the element\_spec method. We continue by displaying each element from the TensorSpec.

**Note:** Along with example and sample, the term element is also used to described a tensor in a dataset.

Simply, we create a dataset containing four elements. Each element contains ten random uniform numbers between 0 and 1. We convert the dataset to one that TensorFlow can consume (or work with). We display each element by iterating the TensorFlow dataset.

# Reading Input Data

If all of your input data fits in memory, the simplest way to create a dataset for TensorFlow consumption is to convert it to a tf.Tensor object with Dataset.from\_tensor\_slices().

# Colab Abends

As previously noted, when running Google Colab for a long time (e.g., several hours) without pause or loading large datasets into memory and processing said data it may crash (or abend). When this happens, you have two choices that we know of:

1. restart all runtimes
2. close the program and restart it from scratch

To restart all runtimes, click Runtime on the top menu, click Restart runtime from the drop-down menu, and click YES when prompted. Google Colab recommends this option. If you restart from scratch, clear browser history first and then start Google Colab from scratch.

# Batch Size

**Batch size** is the number of training examples processed by the neural network model in one pass. Don't get batch size confused with epochs! An **epoch** is a complete pass through the training dataset. So, the number of epochs is the number of complete passes through the training dataset. Epochs have nothing to do with the processing of training examples! They just represent the number of passes through the network. Simply, during each pass (or epoch) through the network, a batch of the training dataset is processed.

The size of a batch must be more than or equal to one and less than or equal to the number of samples in the training dataset. This makes sense because you can't have a batch that is bigger than the total number of training examples.

TensorFlow is optimized to run batches of training data much greater than one. So, a batch size of one is very inefficient! Since a batch size of one represents the entire training dataset, we are not really batching data. So, unless you don't want to batch data, don't use a batch size of one!

# Keras Data

The tf.keras.datasets module provides seven preprocessed datasets for practicing TensorFlow. Peruse the following URL to get more information about the datasets: <https://keras.io/api/datasets/>.

Let’s grab the Keras MNIST dataset:

train, test = tf.keras.datasets.mnist.load\_data(path='mnist.npz')

Both train and test data contain MNIST images and labels in a tuple:

type(train), type(test)

Explore the shape of train data:

print ('train data:', br)

print (train[0].shape)

print (train[1].shape)

Explore the shape of test data:

print ('test data:', br)

print (test[0].shape)

print (test[1].shape)

Train data consists of 60,000 28 x 28 feature images and 60,000 labels. Test data consists of 10,000 28 x 28 feature images and 10,000 labels.

## Build the Input Pipeline

Split train and test data into their respective images and labels. Scale image data. Create TensorFlow consumable data with the from\_tensor\_slices method.

Let’s begin with train data:

train\_images, train\_labels = train

train\_images = train\_images / 255  # divide by 255 to scale

train\_k = tf.data.Dataset.from\_tensor\_slices(

    (train\_images, train\_labels))

train\_k.element\_spec

Continue with test data:

test\_images, test\_labels = test

test\_images = test\_images / 255  # divide by 255 to scale

test\_k = tf.data.Dataset.from\_tensor\_slices(

    (test\_images, test\_labels))

test\_k

Shuffle (where appropriate), batch, and prefetch data:

BATCH\_SIZE = 128

SHUFFLE\_BUFFER\_SIZE = 1000

train\_kd = train\_k.shuffle(

    SHUFFLE\_BUFFER\_SIZE).batch(BATCH\_SIZE).prefetch(1)

test\_kd = test\_k.batch(BATCH\_SIZE).prefetch(1)

## Create the Model

Import libraries:

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense, Flatten, Dropout

from tensorflow import keras

Clear previous models from memory:

# clear any previous models

keras.backend.clear\_session()

Build the model:

model = Sequential([

  Flatten(input\_shape=[28, 28]),

  Dense(512, activation='relu'),

  Dropout(0.5),

  Dense(10, activation='softmax')

])

The model is a feedforward neural network with densely connected layers. That is, all neurons see the data. The first layer accepts 28 x 28 images and flattens each image into a 1D array consisting of 784 pixels. The second layer accepts data into 512 neurons and uses relu activation to minimize loss. The third layer uses dropout to mitigate overfitting. The fourth layer is the output layer. It accepts data into 10 neurons because our data has 10 class labels. It uses softmax activation to reduce loss.

**Dropout** is a regularization technique (patented by Google) for reducing overfitting in neural networks. The technique works by dropping out units in a neural network.

## Model Summary

Display a summary of the model:

model.summary()

Output shape of the first layer is (None, 784). None is the first parameter because TensorFlow models accept any batch size. We get the second parameter of 784 by multiplying 28 by 28 image pixels because we want flattened images. We have 0 parameters because this is the first layer.

Output shape of the second layer is (None, 512). The second parameter is 512 because this is the number of neurons set for this layer. The number of parameters is 401,920 derived by multiplying 512 (neurons at this layer) by 784 (neurons in the previous layer) and adding 512 to account for neurons at this layer.

Output shape of the third layer is (None, 512). Using dropout doesn’t impact the neuron count. So, this layer inherits (None, 512) from the previous layer and has no parameters.

Output shape of the fourth layer is (None, 10). The second parameter is 10 to reflect the number of classes. The number of parameters is 5130 derived by multiplying 10 (neurons at this layer) by 512 (neurons in the previous layer) and adding 10 to account for neurons at this layer.

## Compile the Model

Compile:

model.compile(optimizer='adam',

              loss='sparse\_categorical\_crossentropy',

              metrics=['accuracy'])

## Train the Model

Train:

epochs = 10

history = model.fit(train\_kd, epochs=epochs, verbose=1,

                    validation\_data=test\_kd)

Since epochs is set at 10, our model processes the dataset 10 times. Since train and test accuracy are closely aligned, we don’t have much overfitting.

# Scikit-Learn Data

We can also read data from the scikit-learn library. Scikit-learn is free software machine learning library for the Python programming language.

Let’s grab a dataset from this library:

from sklearn.datasets import fetch\_lfw\_people

faces = fetch\_lfw\_people(min\_faces\_per\_person=70, resize=0.4)

The fetch\_lfw\_people dataset is a collection of JPEG pictures of famous people collected over the internet. All details are available on the official website: <http://vis-www.cs.umass.edu/lfw/>. Each picture is centered on a single face. The typical machine learning task is face verification. So, given a pair of pictures, we predict whether the two images are of the same person.

An alternative machine learning task is face recognition (or face identification). So, given the picture of the face of an unknown person, we identify the name of the person by referring to a gallery of previously seen pictures of identified persons.

## Explore the Data

Display the keys:

# get available keys from dataset

faces.keys()

The dataset contains feature images and targets. It also contains target names and a description of the dataset. The data key points to flattened vectors of each image.

Listing 4-4 displays shapes, target names, and class labels:

image, target = faces.images, faces.target

data = faces.data

names = faces.target\_names

print ('feature image tensor:', br)

print (image.shape, br)

print ('target tensor:', br)

print (target.shape, br)

print ('flattened image tensor:', br)

print (data.shape, br)

print ('target names:', br)

print (names, br)

print ('class labels:', len(names))

Listing 4-4. Information about the dataset

The feature image tensor consists of 1,288 50 x 37 face images. The target tensor consists of 1,288 targets. The data shape consists of 1,288 flattened vectors each with 1,850 elements. We get 1,850 by multiplying 50 by 37.

Listing 4-5 explores the first example:

# display the first data example

i = 0

print ('first image example:', br)

print (image[i], br)

print ('first target example:', br)

print (target[i], br)

print ('name of first target:', br)

print (names[target[i]], br)

print ('first data example (flattened image):', br)

print (data[i], br)

print ('first image:', br)

import matplotlib.pyplot as plt

# display the first image in the dataset

plt.imshow(image[i], cmap='bone')

plt.title(names[target[i]])

Listing 4-5. Explore the first example

Each image is represented by a 2D matrix. The target value for the first image is 5, which just happens to be an image of Hugo Chavez.

## Build the Input Pipeline

Create train and test sets. And, scale feature images as shown in Listing 4-6:

# create train and test data

from sklearn.model\_selection import train\_test\_split

X\_train, X\_test, y\_train, y\_test = train\_test\_split(

    image, target, test\_size=0.33, random\_state=0)

# scale feature image data and create TensorFlow tensors

X\_train = X\_train / 255.0

X\_test = X\_test / 255.0

print ('X\_train shape:', end=' ')

print (X\_train.shape)

print ('X\_test shape:', end=' ')

print (X\_test.shape)

Listing 4-6. Create train and test sets, and scale feature images

The train\_test\_split module provides an easy way to split a dataset into train and test sets. We also have the flexibility to set the train and test size to our needs. The random\_state parameter provides a way to reproduce our results.

Our train set contains 67% of the data. The remaining 33% is placed in the test set. The training feature data shape is (862, 50, 37) and the test feature data shape is (426, 50, 37). So, training feature data consists of 862 50 x 37 pixel images and test feature data consists of 426 50 x 73 pixel images.

Continue by slicing data into TensorFlow consumable pieces:

faces\_train = tf.data.Dataset.from\_tensor\_slices(

    (X\_train, y\_train))

faces\_test = tf.data.Dataset.from\_tensor\_slices(

    (X\_test, y\_test))

Shuffle (where appropriate), batch, and prefetch train and test data:

# prepare tensors for training

BATCH\_SIZE = 16

SHUFFLE\_BUFFER\_SIZE = 100

faces\_train\_ds = faces\_train.shuffle(SHUFFLE\_BUFFER\_SIZE).batch(

    BATCH\_SIZE).prefetch(1)

faces\_test\_ds = faces\_test.batch(BATCH\_SIZE).prefetch(1)

## Build the Model

Build a simple model and train the data as shown in Listing 4-7:

import numpy as np

class\_labels = len(names)

# clear previous model and plant a seed

keras.backend.clear\_session()

np.random.seed(0)

tf.random.set\_seed(0)

model = Sequential([

  Flatten(input\_shape=[50, 37]),

  Dense(16, activation='relu'),

  Dense(class\_labels, activation='softmax')

])

Listing 4-7. Model for faces data

We import the numpy library because we use it to plant a random seed. We create a variable to hold the number of class labels. We plant numpy and TensorFlow random seeds for reproducibility of results. The input shape reflects the image size of feature data.

## Model Summary

Run a model summary:

model.summary()

Output shape of the first layer is (None, 1850). We get 1,850 by multiplying 50 by 37. The number of parameters is 0 because this is the first layer. Output shape of the second layer is (None, 16). We have 16 neurons acting on the data at this layer. The number of parameters is 29,616. We get 29,600 by multiplying 16 by 1,850. We get 29,616 by adding 16 to 29,600 to account for the number of neurons at this layer. Output shape of the third layer is (None, 7). We have 7 neurons at this layer to account for the seven class labels. The number of parameters is 119. We get 112 by multiplying 16 by 7. We get 119 by adding 7 to 112 to account for the number of neurons at this layer. The None parameter is present because TensorFlow accepts any batch size.

## Compile the Model

model.compile(optimizer='adam',

              loss='sparse\_categorical\_crossentropy',

              metrics=['accuracy'])

## Train the Model

history = model.fit(faces\_train\_ds, epochs=10,

                    validation\_data=faces\_test\_ds)

Performance is not good because feedforward nets don’t work well with image data. However, we create this simple model to show you how to train the dataset.

# Numpy Data

We can load numpy data directly and scale it as shown in Listing 4-8:

DATA\_URL = 'https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz'

path = tf.keras.utils.get\_file('mnist.npz', DATA\_URL)

with np.load(path) as data:

  train\_examples = data['x\_train']

  train\_labels = data['y\_train']

  test\_examples = data['x\_test']

  test\_labels = data['y\_test']

train\_scaled = train\_examples / 255.

test\_scaled = test\_examples / 255.

Listing 4-8. Load numpy data from a URL

Identify a URL path to a numpy file. Use tf.keras.utils.get\_file to access the path. Load numpy data with the np.load() function. Split data into train and test sets. Scale feature image data.

## Load numpy arrays with tf.data.Dataset

Convert train and test images and labels to a TensorFlow consumable form:

train\_dataset = tf.data.Dataset.from\_tensor\_slices(

    (train\_scaled, train\_labels))

test\_dataset = tf.data.Dataset.from\_tensor\_slices(

    (test\_scaled, test\_labels))

## Prepare Data for Training

Shuffle (where appropriate), batch, and prefetch data:

BATCH\_SIZE = 128

SHUFFLE\_BUFFER\_SIZE = 1000

train\_np = train\_dataset.shuffle(

    SHUFFLE\_BUFFER\_SIZE).batch(BATCH\_SIZE).prefetch(1)

test\_np = test\_dataset.batch(BATCH\_SIZE).prefetch(1)

## Create the Model

Clear previous sessions and set random seeds:

keras.backend.clear\_session()

np.random.seed(0)

tf.random.set\_seed(0)

Use the same model we used with Keras MNIST:

model = Sequential([

  Flatten(input\_shape=[28, 28]),

  Dense(512, activation='relu'),

  Dropout(0.5),

  Dense(10, activation='softmax')

])

## Model Summary

Since we used the same model, the summary is the same as before:

model.summary()

## Compile the Model

Compile:

model.compile(optimizer='adam',

              loss='sparse\_categorical\_crossentropy',

              metrics=['accuracy'])

## Train the Model

Train:

epochs = 10

history = model.fit(train\_np, epochs=epochs, verbose=1,

                    validation\_data=test\_np)

# CSV Data

A CSV dataset is a comma-separated values file that allows data to be saved in a tabular format. CSV data looks like a conventional spreadsheet but with a .csv extension. CSV files can be used with any spreadsheet program that we know of including Microsoft Excel or Google Spreadsheets.

A great place to get CSV datasets for machine learning is the UCI Machine Learning Repository. The main site for the repository is located at the following URL: https://archive.ics.uci.edu/ml/datasets.php

A CSV dataset commonly cited in machine learning literature is winequality-red.csv. This dataset contains red variants of the Portuguese Vinho Verde wine. For detailed information, consult:

P. Cortez, A. Cerdeira, F. Almeida, T. Matos and J. Reis. Modeling wine preferences by data mining from physicochemical properties. In Decision Support Systems, Elsevier, 2009, 47(4):547-553. ISSN: 0167-9236.

For general information about the dataset, peruse the following URL:

<https://archive.ics.uci.edu/ml/datasets/wine+quality>

## Dataset Characteristics

The dataset consists of eleven independent feature variables and one target variable. Feature variables include:

* fixed acidity
* volatile acidity
* citric acid
* residual sugar
* chlorides
* free sulfur dioxide
* total sulfur dioxide
* density
* pH
* sulphates
* alcohol

The target variable is:

* quality

The target variable (quality) can take on a score between 0 and 10. A score of 0 means that the quality is very low, whereas a score of 10 means that the quality is very high. The dataset contains 1599 examples.

## Get Data

Identify the dataset URL:

url = 'http://archive.ics.uci.edu/ml/machine-learning-databases/wine-quality/winequality-red.csv'

Establish the path to the dataset:

path = keras.utils.get\_file('winequality-red.csv', url)

path

Create a pandas DataFrame from the CSV file and place it into a Python variable:

import pandas as pd

data = pd.read\_csv(path, sep = ';')

View records from the beginning of the DataFrame:

data.head()

View records from the end of the DataFrame:

data.tail()

Identify class labels used in the dataset:

data.quality.unique()

Since quality is the target, we use the unique() function to pull distinct labels from the dataset. The dataset contains scores 3, 4, 5, 6, 7, and 8. However, quality can take on scores between 0 and 10, so there are eleven possible class labels.

Display datatypes:

data.dtypes

Features are datatype float64 and the target is datatype int64.

Display number of examples in the dataset:

len(data)

We have 1,599 examples.

## Prepare Data for TensorFlow Consumption

Create the target set:

# create a copy of the DataFrame

df = data.copy()

# create the target

target = df.pop('quality')

Create a copy of the DataFrame to preserve the original data. Pop the target column into a variable to create the target dataset. The pop method removes the column permanently.

Show what happened to the DataFrame copy:

df.head()

Notice that the quality column is no longer in the DataFrame.

Convert the DataFrame to numpy values:

features = df.values

labels = target.values

Split into train and test sets, and scale feature data as shown in Listing 4-9:

X\_train, X\_test, y\_train, y\_test = train\_test\_split(

    features, labels, test\_size=0.33, random\_state=0)

# scale feature image data and create TensorFlow tensors

from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()

X\_train\_std = scaler.fit\_transform(X\_train)

X\_test\_std = scaler.fit\_transform(X\_test)

Listing 4-9. Split data into train and test sets, and scale feature data

Feature data are not images. They are scalar values. So, we use the StandardScaler module to transform continuous feature data to mean of 0 and standard deviation of 1 before applying machine learning techniques.

## Prepare Data for TensorFlow Consumption

Slice train and test sets into tf.Data.Dataset data:

train\_wine = tf.data.Dataset.from\_tensor\_slices(

    (X\_train\_std, y\_train))

test\_wine = tf.data.Dataset.from\_tensor\_slices(

    (X\_test\_std, y\_test))

Create a function to view tensors:

def see\_samples(data, num):

  for feat, targ in data.take(num):

    print ('Features: {}'.format(feat))

    print ('Target: {}'.format(targ), br)

View the first three tensors:

n = 3

see\_samples(train\_wine, n)

Shuffle (where appropriate), batch, and prefetch data:

BATCH\_SIZE = 16

SHUFFLE\_BUFFER\_SIZE = 100

train\_wine\_ds = train\_wine.shuffle(

    SHUFFLE\_BUFFER\_SIZE).batch(BATCH\_SIZE).prefetch(1)

test\_wine\_ds = test\_wine.batch(BATCH\_SIZE).prefetch(1)

train\_wine\_ds, test\_wine\_ds

## Build the Model

Clear sessions and set random seeds:

keras.backend.clear\_session()

np.random.seed(0)

tf.random.set\_seed(0)

Create the model:

model = Sequential([

  Dense(30, activation='relu', input\_shape=[11,]),

  Dense(11, activation='softmax')

])

The input shape is [11,] because the dataset has eleven features. The output shape is eleven because the dataset has eleven class labels.

## Model Summary

model.summary()

Output shape of the first layer is (None, 30) because we have 30 neurons at this layer. Number of parameters is 360. We get 330 by multiplying 30 neurons by 11 features. We get 360 by adding 30 neurons at this layer to 330. Output shape at the second layer is (None, 11) because we have 11 neurons at this layer to account for the number of possible labels. Number of parameters is 341. We get 330 by multiplying 30 neurons from the previous layer by 11 neurons at this layer. We get 341 by adding 11 neurons at this layer to 330.

## Compile the Model

model.compile(optimizer='adam',

              loss='sparse\_categorical\_crossentropy',

              metrics=['accuracy'])

## Train the Model

history = model.fit(train\_wine\_ds, epochs=10,

                    validation\_data=test\_wine\_ds)

We don’t get very good performance, but we just wanted to show you how to train a CSV dataset.

# Data Datasets

Data with the .data extension is not CSV data. Since this type of data is not CSV, we cannot use tf.keras.utils.get\_file. Instead, we download the dataset, copy it to Google Drive, and access it from Colab.

# Abalone Dataset

The abalone dataset is type DATA. The general site for the data is:

<https://archive.ics.uci.edu/ml/datasets/Abalone>

To download Abalone data, go to the URL, click on Data Folder, and click on abalone.data. The dataset is automatically downloaded to your Downloads directory. Since we are working with the Colab cloud service, we copy the file to the Colab Notebooks directory on your Google Drive. The easiest way to copy is to drag and drop the file from your Downloads directory into Google Drive.

The other file of interest is abalone.names, which provides an in-depth description of the dataset. Like the data file, it can be accessed from the Data Folder by clicking on abalone.names.

## Dataset Characteristics

The dataset is used for predicting age of abalone shells from physical measurements. The age of an abalone shell is partially determined by cutting through the cone, staining it, and counting the number of rings through a microscope. Other measurements supplement age prediction.

Feature variables include:

* sex
* length
* diameter
* height
* whole
* shucked
* viscera
* shell

The target variable is:

* rings

The target variable (rings) can take on a score between 1 and 29. Such scores represent the number of rings for an abalone shell. So, the number of rings is the value to predict. An interesting notion about this dataset is that we can use it either as a continuous value experiment or as a classification problem.

## Mount Google Drive to Colab

To enable us to work with the dataset, we mount Colab to Google Drive:

from google.colab import drive

drive.mount('/content/gdrive')

Click on the URL, choose a Google account, click Allow, copy the authorization code, and press the Enter key.

**Note:** Be sure to copy the file to the Colab Notebooks directory on Google Drive!

Establish the path in Colab:

# establish path (be sure to copy file to Google Drive)

path = 'gdrive/My Drive/Colab Notebooks/'

abalone = path + 'abalone.data'

abalone

## Read Data

Since the dataset doesn’t contain column headings, we need to define them before reading the dataset:

cols = ['Sex', 'Length', 'Diameter', 'Height', 'Whole',

        'Shucked', 'Viscera', 'Shell', 'Rings']

ab\_data = pd.read\_csv(abalone, names=cols)

## Explore Data

Display records from the beginning of the dataset:

ab\_data.head(3)

Display records from the end of the dataset:

ab\_data.tail(3)

Return number of records:

len(ab\_data)

So, we have 4,177 records.

Display the output classes used in the dataset:

# classes used

print ('classes:', br)

print (np.sort(ab\_data['Rings'].unique()))

Display number of output classes:

# number of classes

print ('number of classes:', len(ab\_data['Rings'].unique()))

We have 28 classes.

Display class distribution:

instance = ab\_data['Rings'].value\_counts()

instance.to\_dict()

Classes range from 1 to 27, and 29. Each class represents the age of an abalone shell in years. The distribution is very uneven. For example, we have 689 instances of shells that are nine years old, but only 1 instance of a shell that is a one-year-old. Machine learning algorithms don’t work well with unbalanced data because prediction is biased towards the classes with the most instances. **Unbalanced data** is one with unequal instances for different classes.

Display datatypes:

ab\_data.dtypes

Features, with the exception of sex, are float64. The target is int64.

Display information about all columns:

ab\_data.info(verbose=True)

Display shape:

ab\_data.shape

We have 4,177 examples with 9 attributes each.

## Create Train and Test Sets

Split data:

train, test = train\_test\_split(ab\_data)

print(len(train), 'train examples')

print(len(test), 'test examples')

We have 3,132 train examples and 1,045 test examples. If not specified, the default test size is 25%.

## Create Feature and Target Sets

It is a good idea to create copies of train and test sets to preserve original data. Otherwise the pop method can wreak havoc as it removes data permanently.

Create targets:

train\_copy, test\_copy = train.copy(), test.copy()

# create targets

train\_target, test\_target = train\_copy.pop('Rings'),\

test\_copy.pop('Rings')

Verify targets:

len(train\_target), len(test\_target)

Verify train feature data:

train\_copy.head(3)

When wrangling data, it is a good idea to verify contents.

Convert feature data to numpy:

train\_features, test\_features = train\_copy.values,\

test\_copy.values

## Scale Features

We can only scale continuous values. Since the sex feature is not continuous, it cannot be scaled. So, we slice off the continuous values to scale. We then recreate train and test sets with the sex feature and scaled continuous values.

Display a sample to verify slicing:

train\_features[0], test\_features[0]

Slicing is complex. So, it is a good idea to display a sample to ensure that slicing worked as expected.

Create two train sets (one with sex and the other with continuous values):

train\_sex = [row[0] for row in train\_features]

train\_f = [row[1:] for row in train\_features]

train\_sex[0], train\_f[0]

So far so good!

Create two test sets (one with sex and the other with continuous values):

test\_sex = [row[0] for row in test\_features]

test\_f = [row[1:] for row in test\_features]

test\_sex[0], test\_f[0]

Again, our slices match the original data.

Scale continuous values:

scaler = StandardScaler()

train\_sc = scaler.fit\_transform(train\_f)

test\_sc = scaler.fit\_transform(test\_f)

## Create Train and Test Sets with Sex and Scaled Values

Now that we’ve scaled continuous values, we need to recombine it with the sex feature as shown in Listing 4-10:

train\_ds = [np.append(train\_sex[i], row)

for i, row in enumerate(train\_sc)]

test\_ds = [np.append(test\_sex[i], row)

for i, row in enumerate(test\_sc)]

train\_ds[0], test\_ds[0]

Listing 4-10. Recombine continuous features with the sex feature

Recombine train and test features and display. Notice that the continuous features are scaled.

## Convert Numpy Feature Sets into Pandas DataFrames

To properly build a TensorFlow consumable dataset with non-continuous data, we need the feature data in pandas DataFrame form:

col = ['Sex', 'Length', 'Diameter', 'Height', 'Whole',

       'Shucked', 'Viscera', 'Shell']

train\_ab = pd.DataFrame(train\_ds, columns=col)

test\_ab = pd.DataFrame(test\_ds, columns=col)

We need the original column names to add to the DataFrame.

Verify train features:

train\_ab.tail(3)

Verify test features:

test\_ab.tail(3)

## Build the Input Pipeline

Prepare train and test data for TensorFlow consumption:

train\_ds = tf.data.Dataset.from\_tensor\_slices(

    (dict(train\_ab), train\_target))

test\_ds = tf.data.Dataset.from\_tensor\_slices(

    (dict(test\_ab), test\_target))

Notice that we convert train and test feature data to Python dictionaries. We do this to enable construction of categorical feature columns.

Shuffle (where appropriate), batch, and prefetch data:

BATCH\_SIZE = 32

SHUFFLE\_BUFFER\_SIZE = 100

train\_ads = train\_ds.shuffle(

    SHUFFLE\_BUFFER\_SIZE).batch(BATCH\_SIZE).prefetch(1)

test\_ads = test\_ds.batch(BATCH\_SIZE).prefetch(1)

train\_ads, test\_ads

Notice that the shapes include each feature column name.

## Explore a Batch

Since we converted feature data to dictionaries, we can display interesting information about the data as shown in Listing 4-11:

def see\_format(data, num, feature, indx):

  for feature\_batch, label\_batch in data.take(num):

    print('Every feature:', list(feature\_batch.keys()))

    print('One example from a batch of ' + feature + ':',

          feature\_batch[feature][indx])

    print('One example from a batch of targets:',

          label\_batch[indx])

print ('train sample:')

see\_format(train\_ads, 1, 'Height', 0)

print ()

print ('test sample:')

see\_format(test\_ads, 1, 'Sex', 0)

Listing 4-11. Display information about a sample batch

## Categorical Columns

TensorFlow consumption is limited to numeric data. So, we must convert any categorical data. The only culprit in this scenario is the 'Sex' feature because it is represented by string values of either 'M', 'F' or 'I'. So, sex for the abalone shell is either male, female or infant.

Since we cannot feed strings directly to a model, we must first map them to numeric values. The categorical vocabulary columns feature provides a way to represent strings as a one-hot vector. This process is called **one-hot encoding**, which is a technique that converts categorical variables to numerical in an interpretable format.

The strings are converted in the following manner:

1. 'M' => 1 0 0
2. 'F' => 0 1 0
3. 'I' => 0 0 1

Listing 4-12 one-hot encodes the sex feature:

from tensorflow import feature\_column

sex\_one\_hot =\

feature\_column.categorical\_column\_with\_vocabulary\_list(

    'Sex', ['M', 'F', 'I'])

print (type(sex\_one\_hot))

feature\_columns =\

[tf.feature\_column.indicator\_column(sex\_one\_hot)]

feature\_layer = tf.keras.layers.DenseFeatures(feature\_columns)

Listing 4-12. One-hot encode the sex feature

Import the feature\_column module. One-hot encode 'Sex'. Create the feature\_columns list. We create a list so we can have multiple categorical features in a training dataset. Finally, create the feature\_layer for the model.

For a comprehensive example, peruse the following URL:

<https://www.tensorflow.org/tutorials/structured_data/feature_columns>

## Build the Model

# clear previous model and plant a seed

keras.backend.clear\_session()

model = tf.keras.Sequential([

  feature\_layer,

  Dense(128, activation='relu'),

  Dense(128, activation='relu'),

  Dense(29, activation='sigmoid')

])

Notice that the first layer is feature\_layer, which informs the model about one-hot encoded features.

## Compile the Model

model.compile(optimizer='adam',

              loss='sparse\_categorical\_crossentropy',

              metrics=['accuracy'])

## Train the Model

model.fit(train\_ads,

          validation\_data=test\_ads,

          epochs=1)

We only trained for 1 epoch because we knew that performance would be horrible. How did we know this? Check out the next section to find out.

## Unbalanced and Irregular Data

The Abalone dataset is not a good dataset to make predictions for two reasons:

1. the dataset is unbalanced
2. the dataset is irregular

An **unbalanced dataset** is one where the classes are not represented equally. That is, classes don't have the same number of examples. This dataset is especially unbalanced because some classes have one example while others have hundreds of examples. Training with an unbalanced dataset won't produce good results. So, we won't learn much. The reason is that predictions are biased toward classes with more instances!

An **irregular dataset** is one with too many target (or label) classes, but not enough data. We should always check the number of samples (or examples) per label in our dataset. A class label with not enough samples is harder to learn from.