3

Working with TensorFlow Data

We introduce TensorFlow Datasets (TFDS). We discuss many facets of a TFDS with code examples. We continue with a complete TFDS modeling example. Notebooks for this chapter are located at the following URL: <https://github.com/paperd/tensorflow>.

TFDS is a collection of datasets ready to use with TensorFlow. Like all TensorFlow consumable datasets, TFDS are exposed as tf.data.Datasets, which allows us to create easy-to-use and high performance input pipelines.

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.

Let’s begin by displaying the current version of TensorFlow and enabling 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.

# TensorFlow Datasets (TFDS)

The following URLs provide more information about TFDS:

* <https://www.tensorflow.org/datasets>
* <https://www.tensorflow.org/datasets/overview>
* <https://www.tensorflow.org/datasets/catalog/overview>

Start with the first URL as it introduces TFDS. The second URL demonstrates how to display a list of all TFDS and additional technical information. The third URL shows how TFDS are categorized.

# Colab Abends

When we run Google Colab for a long time (several hours) without pause or load large datasets into memory and process 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

# Available TFDS

Let’s begin by displaying a list of available TFDS:

import tensorflow\_datasets as tfds

# See available datasets

tfds.list\_builders()

Begin by importing the tfds module. Use the list\_builders() method to display.

Find out how many TFDS are in the tensorflow\_datasets container:

print (str(len(tfds.list\_builders())) + ' datasets')

Wow! There are 155 datasets we can use to practice TensorFlow.

Peruse the following URL for a really nice tutorial on TFDS:

<https://colab.research.google.com/github/tensorflow/datasets/blob/master/docs/overview.ipynb>

# Load a TFDS

We can load a TFDS with one line of code! In the tutorial on the TFDS website, it states that we must install tensorflow-datasets. But, in Google Colab, we don’t have to do this.

**Tip:** If you are working in an environment other than Google Colab, you can install the TFDS module by running the following code snippet: !pip install tensorflow-datasets

Import the tfds module:

import tensorflow\_datasets as tfds

Load training data directly with tfds.load():

# load train set

train, info = tfds.load('mnist', split='train', with\_info=True)

info

The **tfds.load()** function loads that named dataset into a tf.data.Dataset. We add the info element to enable display of useful metadata about the dataset. **Metadata** is a set of data that describes and gives information about other data.

Load test data directly:

# load test data

test, info = tfds.load('mnist', split='test', with\_info=True)

info

We loaded the MNIST train and test dataset from the TFDS container. The **MNIST** database (Modified National Institute of Standards and Technology database) is a large database of handwritten digits that is commonly used for training various image processing systems. The database is also widely used for training and testing in the field of machine learning. The database consists of a set of 60,000 training examples, and 10,000 test examples. We included the info element, which provides detailed information about the dataset.

**Note:** In machine learning vernacular, a data element is typically described as either an example or sample. The word example is used interchangeably with sample.

Although feed-forward neural nets don't tend to perform well on images, MNIST is an exception because it is heavily preprocessed. That is, MNIST images are roughly of the same size, centered in the middle of the image space, and vertically oriented.

# Extract Useful Information

The info element includes methods that allow us to extract specific information about a dataset.

Listing 3-1 extracts information about the classes:

# create a variable to hold a return symbol

br = '\n'

# display number of classes

num\_classes = info.features['label'].num\_classes

class\_labels = info.features['label'].names

# display class labels

print ('number of classes:', num\_classes)

print ('class labels:', class\_labels)

Listing 3-1. Extracting meaningful information from a dataset

We just extracted the number of classes and class labels with the features method.

# Inspect the TFDS

We have two ways to inspect an element:

1. Print the element
2. Print the element with the element\_spec method

Print elements:

# display training and test set

print (train)

print (test)

Print elements with element\_spec:

# display with element\_spec method

print (train.element\_spec)

print (test.element\_spec)

Either way, the output is very similar. Tensor shapes are (28, 28, 1). So, training and test data consists of 28 x 28 images. The 1 value means that images are displayed in grayscale. Image data (the feature set) is composed of tf.uint8 data and label (or target) data is composed of tf.int64 data.

A grayscale image is one where the value of each pixel is a single sample representing only an amount of light. That is, it carries only intensity information. Grayscale images are composed exclusively of shades of gray. The contrast of an image ranges from black at the weakest intensity to white at the strongest.

We can also display training examples from a TFDS with one line of code:

# Show train feature image examples

fig = tfds.show\_examples(info, train)

The show\_examples method displays sample images from a tf.data.Dataset.

# Feature Dictionaries

All TFDS contain feature dictionaries that map feature names to Tensor values. By default, tfds.load returns a dictionary of tf.Tensors. A **tf.Tensor** represents a rectangular array of data in TensorFlow.

A typical dataset, like MNIST, has 2 keys: image and label. Let’s inspect one sample with take(1). The number we feed into the take() function renders the number of samples we receive from the dataset.

Take one sample from the train dataset and display its keys:

for sample in train.take(1):

  print (list(sample.keys()))

We see the two keys as expected. The image key references the images in the dataset. The label key references the labels in the dataset. The formal dictionary structure is represented as: {'image': tf.Tensor, 'label': tf.Tensor}.

Now that we know the keys, we can easily display the feature shape and target value from the first train sample:

for sample in train.take(1):

  print ('feature shape:', sample['image'].shape)

  print ('target value: ', sample['label'].numpy())

The shape of the first feature sample is (28, 28, 1) and the value of the first label is 4. We used the numpy() method to convert the target tensor to a scalar value. Since any dataset consumed by a machine learning algorithm must have the **same shape**, we get the shape of a dataset from a single sample!

Let’s get nine examples from the train set:

n, ls = 9, []

for sample in train.take(n):

  ls.append(sample['label'].numpy())

ls

We see [4, 1, 0, 7, 8, 1, 2, 7, 1], which matches the labels from show\_examples in the previous section.

By using the as\_supervised=True parameter with tfds.load, we get a tuple of (features, label) instead of a dictionary:

ds = tfds.load('mnist', split='train', as\_supervised=True)

ds = ds.take(1)

for image, label in ds:

  print (image.shape, br, label)

The sample is in the form (image, label).

We can also get a numpy tuple of (features, label):

ds = tfds.load('mnist', split='train', as\_supervised=True)

ds = ds.take(1)

for image, label in tfds.as\_numpy(ds):

  print (type(image), type(label), label)

We use tfds.as\_numpy to convert tf.Tensor to np.array and tf.data.Dataset to Generator[np.array].

Finally, we can get a batched tf.Tensor:

image, label = tfds.as\_numpy(tfds.load(

    'mnist',

    split='train',

    batch\_size=-1,

    as\_supervised=True,

))

type(image), image.shape

By using batch\_size=-1 we can load the full dataset in a single batch. We see numpy tensor (60000, 28, 28, 1), which means training data contains 60,000 28 x 28 grayscale images.

In summary, tfds.load() returns a dictionary by default, a tuple with as\_supervised=True of tf.Tensor or a np.array with tfds.as\_numpy. Be careful that your dataset can fit in memory and that all examples have the same shape.

# Build the Input Pipeline

Shuffle and batch train data:

train\_ds = train.shuffle(10000).batch(32)

Scale images:

train\_ds = train\_ds.map(lambda items:\

                        (tf.cast(items['image'],\

                                 tf.float32) / 255.,\

                         items['label']))

Although this looks complicated, we use a lambda function to divide each image by 255.

Prefetch train data:

train\_ds = train\_ds.prefetch(1)

**Prefetching** improves model performance because it adds efficiency to the batching process. While our training algorithm is working on one batch, TensorFlow is working on the dataset in parallel to get the next batch ready.

Batch test data:

test\_ds = test.batch(10000)

We don’t shuffle test data because it is new data.

Scale images:

test\_ds = test\_ds.map(lambda items:\

                      (tf.cast(items['image'],\

                               tf.float32) / 255.,\

                       items['label']))

Prefetch test data:

test\_ds = test\_ds.prefetch(1)

# Build the Model

Import libraries:

from tensorflow.keras.models import Sequential

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

from tensorflow import keras

Create the model:

# clear previous model

keras.backend.clear\_session()

model = Sequential([

  Flatten(input\_shape=[28, 28, 1]),

  Dense(512, activation='relu'),

  Dense(10, activation='softmax')

])

The model has an input layer, a dense hidden layer, and a dense output layer. The input layer flattens images for processing in the next layer. The hidden layer accepts data into 512 neurons. The output layer accepts data from the hidden layer into 10 neurons that represent the ten digit classes.

# Model Summary

Display a summary of the model:

model.summary()

The first layer accepts 28 x 28 grayscale images. So, we get Output Shape (None, 784). None is the first parameter because TensorFlow models can accept any batch size. We get the second parameter 784 by multiplying 28 by 28 by 1. So, each image has 784 pixels. The number of parameters is 0 because this is the first layer.

The second layer Output Shape is (None, 512) since we have 512 neurons. The number of parameters is 401920. We get 401408 by multiplying 784 neurons from the first layer by 512 neurons in this layer. We then add 512 neurons from this layer to get 401920.

The third layer Output Shape is (None, 10) since we have ten classes. We get 5130 by multiplying 10 by 512 neurons from the second layer, and adding 10 neurons from this layer.

# Compile the Model

Compile the model with optimizer, loss, and metrics parameters:

model.compile(optimizer='adam',

              loss='sparse\_categorical\_crossentropy',

              metrics=['accuracy'])

Train the model:

epochs = 3

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

                     validation\_data=test\_ds)

We train the model for 3 epochs. That is, we pass data three times through the model. We validate the model with test data. We get pretty good results with not much overfitting with just 3 epochs!

# Generalize on Test Data

It’s always a good idea to evaluate based on test data:

model.evaluate(test\_ds)

# Visualize Performance

Get the training record into a variable:

# get training record into a variable

history\_dict = history.history

Listing 3-2 plots accuracy and loss for the model:

import matplotlib.pyplot as plt

acc = history\_dict['accuracy']

val\_acc = history\_dict['val\_accuracy']

loss = history\_dict['loss']

val\_loss = history\_dict['val\_loss']

epochs = range(1, len(acc) + 1)

plt.figure(figsize=(12,9))

plt.plot(epochs, acc, 'bo', label='Training acc')

plt.plot(epochs, val\_acc, 'b', label='Validation acc')

plt.title('Training and validation accuracy')

plt.xlabel('Epochs')

plt.ylabel('Accuracy')

plt.legend(loc='lower right')

plt.ylim((0.5,1))

plt.show()

# clear previous figure

plt.clf()

plt.figure(figsize=(12,9))

plt.plot(epochs, loss, 'bo', label='Training loss')

plt.plot(epochs, val\_loss, 'b', label='Validation loss')

plt.title('Training and validation loss')

plt.xlabel('Epochs')

plt.ylabel('Loss')

plt.legend()

plt.show()

Listing 3-2. Visualize training performance

The visualization shows that our model fits data pretty well because train and test accuracy are very well aligned!

# DatasetBuilder (tfds.builder)

Since tfds.load is really just a thin convenience wrapper around DatasetBuilder, we can build an input pipeline with the MNIST dataset directly with tfds.builder. **DatasetBuilder** is an abstract base class for all datasets. That is, every TensorFlow dataset is exposed as a DatasetBuilder.

DatasetBuilder performs several duties for TensorFlow datasets:

* where to download data from and how to extract it and write it to a standard format with DatasetBuilder.download\_and\_prepare
* how to load data from disk with DatasetBuilder.as\_dataset
* all information about data including names, types, feature shapes, number of records in each train and test split, and source URLs with DatasetBuilder.info
* ability to directly instantiate any DatasetBuilder with tfds.builder

Unlike tfds.load, however, we have to manually fetch the DatasetBuilder by name, call download\_and\_prepare(), and call as\_dataset(). The advantage of tfds.builder() is that it allows more control over the loading process should we need it.

Listing 3-3 shows how to load MNIST with tfds.builder:

mnist\_builder = tfds.builder('mnist')

mnist\_info = mnist\_builder.info

mnist\_builder.download\_and\_prepare()

datasets = mnist\_builder.as\_dataset()

Listing 3-3. Load MNIST with tfds.builder

Begin with tfds.builder to create the dataset. Include information with the info method. Process data with the download\_and\_prepare method. Place the processed dataset into a variable.

Build the train and test sets:

mnist\_train, mnist\_test = datasets['train'], datasets['test']

Use the feature dictionary to get critical information:

for sample in mnist\_train.take(1):

  print ('feature shape:', sample['image'].shape)

  print ('target value: ', sample['label'].numpy())

We see that the first feature has a shape of (28, 28, 1) and target value of 4.

# MNIST Metadata

Like tfds.load, tfds.builder can access metadata about MNIST:

mnist\_info

We see a lot of useful information about MNIST.

Access feature information:

mnist\_info.features

We see useful information about images and labels.

Listing 3-4 displays the number of classes and class labels:

# create a variable to hold a return symbol

br = '\n'

# display number of classes

num\_classes = mnist\_info.features['label'].num\_classes

class\_labels = mnist\_info.features['label'].names

# display class labels

print ('number of classes:', num\_classes)

print ('class labels:', class\_labels)

Listing 3-4. Number of classes and class labels

Access shapes and datatypes:

print (mnist\_info.features.shape)

print (mnist\_info.features.dtype)

Access image information:

print (mnist\_info.features['image'].shape)

print (mnist\_info.features['image'].dtype)

Access label information:

print (mnist\_info.features['label'].shape)

print (mnist\_info.features['label'].dtype)

Train and test splits:

print (mnist\_info.splits)

Available split keys:

print (list(mnist\_info.splits.keys()))

Number of train and test examples:

print (mnist\_info.splits['train'].num\_examples)

print (mnist\_info.splits['test'].num\_examples)

**Note:** tfds.load has access to the same metadata as tfds.builder.

# Show Examples

As demonstrated earlier in the chapter, tfds.show\_examples allows us to conveniently visualize images (and labels) from an image classification dataset.

Let’s show examples from the test set:

fig = tfds.show\_examples(info, mnist\_test)

# Prepare DatasetBuilder Data

Prepare the input pipeline for DatasetBuilder train and test data.

Shuffle and batch train data:

train\_build = mnist\_train.shuffle(1024).batch(128)

Scale train data:

train\_build = train\_build.map(lambda items:\

                              (tf.cast(items['image'],\

                                       tf.float32) / 255.,\

                               items['label']))

Prefetch train data:

train\_build.prefetch(1)

Batch test data:

test\_build = mnist\_test.batch(128)

Scale test data:

test\_build = test\_build.map(lambda items:\

                            (tf.cast(items['image'],\

                                     tf.float32) / 255.,\

                             items['label']))

Prefetch test data:

test\_build = test\_build.prefetch(1)

# Build the Model

Create the model:

keras.backend.clear\_session()

model = Sequential([

  Flatten(input\_shape=[28, 28, 1]),

  Dense(512, activation='relu'),

  Dense(10, activation='softmax')

])

# Compile the Model

Compile:

model.compile(loss='sparse\_categorical\_crossentropy',

              optimizer='adam',

              metrics=['accuracy'])

# Train the Model

Train:

model.fit(train\_build, epochs=3, validation\_data=test\_build)

As expected, results are very similar to training with tfds.load.

# Load CIFAR-10

Let's use the DatasetBuilder method tfds.builder to manipulate another dataset and show some examples. The **CIFAR-10** dataset consists of 60,000 32 x 32 color images in 10 classes, with 6000 images per class. There are 50,000 training images and 10,000 test images.

The ten classes are:

[airplane, automobile, bird, cat, deer, dog, frog, horse, ship, truck]

The classes are completely mutually exclusive. There is no overlap between automobiles and trucks. Class automobile includes sedans, SUVs, and other items of that sort. Class truck includes only big trucks. Neither includes pickup trucks.

The dataset is divided into five training batches and one test batch, each with 10,000 images. The test batch contains exactly 1,000 randomly-selected images from each class. Training batches contain the remaining images in random order, but some training batches may contain more images from one class than another. Between them, training batches contain exactly 5,000 images from each class. Since CIFAR10 is preprocessed with batches, we can use this information to better batch data for the training model.

Listing 3-5 loads and processes the dataset for TensorFlow consumption:

cifar10\_builder = tfds.builder('cifar10')

cifar10\_info = cifar10\_builder.info

cifar10\_builder.download\_and\_prepare()

cifar10\_train = cifar10\_builder.as\_dataset(split='train')

cifar10\_test = cifar10\_builder.as\_dataset(split='test')

Listing 3-5. Prepare the CIFAR10 dataset for TensorFlow consumption

Inspect the train set:

cifar10\_train

We see that image tensors have shape (32, 32, 3). And, label tensors have shape (). So, each image is represented by 32 x 32 pixels. The 3 value means that images are displayed in color. Image tensors are datatype tf.uint8. Each label is a scalar value. Label tensors are datatype tf.int64.

TensorsFlow leverages the RGB color model to produce color images. The **RGB color model** is an additive color model where red, green, and blue light are added together in various ways to reproduce a broad array of colors. The name of the model comes from the initials of the three additive primary colors, namely, red, green, and blue.

# Inspect the Dataset

Get information about the dataset:

cifar10\_info

Access feature information:

cifar10\_info.features

Get class names:

cifar10\_info.features['label'].names

Get available split keys:

print (list(cifar10\_info.splits.keys()))

Show train examples:

fig = tfds.show\_examples(info, cifar10\_train)

Use the feature dictionary to display train labels:

[sample['label'].numpy() for sample in cifar10\_train.take(9)]

To simplify coding, we used list comprehension.

For completeness, show test examples:

fig = tfds.show\_examples(info, cifar10\_test)

# Prepare the Input Pipeline

Shuffle and batch train data:

train\_cd = cifar10\_train.shuffle(1024).batch(128)

Scale train data:

train\_cd = train\_cd.map(lambda items:\

                        (tf.cast(items['image'],\

                                 tf.float32) / 255.,\

                         items['label']))

Prefetch train data:

train\_cd.prefetch(1)

Batch test data:

test\_cd = cifar10\_test.batch(128)

Scale test data:

test\_cd = test\_cd.map(lambda items:\

                      (tf.cast(items['image'],\

                               tf.float32) / 255.,\

                       items['label']))

Prefetch test data:

test\_cd.prefetch(1)

Build the Model

keras.backend.clear\_session()

model = Sequential([

  Flatten(input\_shape=[32, 32, 3]),

  Dense(512, activation='relu'),

  Dense(10, activation='softmax')

])

We must get the input shape correct. For CIFAR10, input shape is (32, 32, 3)!

# Model Summary

model.summary()

The first layer accepts 32 x 32 color images. So, we get Output Shape (None, 3072). None is the first parameter because TensorFlow models can accept any batch size. We get the second parameter 3072 by multiplying 32 by 32 by 3. Each image has 1,024 pixels, which is the result of multiplying 32 by 32. Since each image is in color, we multiply 1,024 by 3 to get 3,072 neurons. The number of parameters is 0 because this is the first layer.

The second layer Output Shape is (None, 512) since we have 512 neurons. The number of parameters is 1573376. We get 1572864 by multiplying 3,072 neurons from the first layer by 512 neurons in this layer. We then add 512 neurons from this layer to get 1,573,376.

The third layer Output Shape is (None, 10) since we have ten classes. We get 5130 by multiplying 10 by 512 neurons from the second layer, and adding 10 neurons from this layer.

# Compile the Model

model.compile(optimizer='adam',

              loss='sparse\_categorical\_crossentropy',

              metrics=['accuracy'])

# Train the Model

epochs = 3

history = model.fit(train\_cd, epochs=epochs, verbose=1, validation\_data=test\_cd)

Accuracy below 50% is worse than guessing. The probability of guessing heads or tails when flipping a coin if 50%. Our model performed very poorly because feedforward neural networks are not designed to work well with image data.

Our goal was to show you how to load and model a TFDS. We introduce models that work well with image data in a later chapter.