How to Load CSV and Numpy File Types in TensorFlow 2.0

Learning Objectives

1. Load a CSV file into a `tf.data.Dataset`.
2. Load Numpy data

Introduction

In this lab, you load CSV data from a file into a `tf.data.Dataset`. This tutorial provides an example of loading data from NumPy arrays into a `tf.data.Dataset` you also load text data.

Each learning objective will correspond to a **#TODO** in the student lab notebook -- try to complete that notebook first before reviewing this solution notebook.

Load necessary libraries

We will start by importing the necessary libraries for this lab.

In [1]:
```
# You can use any Python source file as a module by executing an import statement in so
# The import statement combines two operations; it searches for the named module, then
# results of that search to a name in the local scope.
import functools

import numpy as np
import tensorflow as tf

print("TensorFlow version: ",tf.version.VERSION)
```

```
TensorFlow version:  2.3.0-dev20200613
```

In [2]:
```
TRAIN_DATA_URL = "https://storage.googleapis.com/tf-datasets/titanic/train.csv"
TEST_DATA_URL = "https://storage.googleapis.com/tf-datasets/titanic/eval.csv"

# Downloads a file from a URL if it not already in the cache using `tf.keras.utils.get_
train_file_path = tf.keras.utils.get_file("train.csv", TRAIN_DATA_URL)
test_file_path = tf.keras.utils.get_file("eval.csv", TEST_DATA_URL)
```

In [3]:
```
# Make numpy values easier to read.
np.set_printoptions(precision=3, suppress=True)
```

Load data

This section provides an example of how to load CSV data from a file into a `tf.data.Dataset`. The data used in this tutorial are taken from the Titanic passenger list. The model will predict the

likelihood a passenger survived based on characteristics like age, gender, ticket class, and whether the person was traveling alone.

To start, let's look at the top of the CSV file to see how it is formatted.

In [4]:
```
# `head()` function is used to get the first n rows
!head {train_file_path}
```

```
survived,sex,age,n_siblings_spouses,parch,fare,class,deck,embark_town,alone
0,male,22.0,1,0,7.25,Third,unknown,Southampton,n
1,female,38.0,1,0,71.2833,First,C,Cherbourg,n
1,female,26.0,0,0,7.925,Third,unknown,Southampton,y
1,female,35.0,1,0,53.1,First,C,Southampton,n
0,male,28.0,0,0,8.4583,Third,unknown,Queenstown,y
0,male,2.0,3,1,21.075,Third,unknown,Southampton,n
1,female,27.0,0,2,11.1333,Third,unknown,Southampton,n
1,female,14.0,1,0,30.0708,Second,unknown,Cherbourg,n
1,female,4.0,1,1,16.7,Third,G,Southampton,n
```

You can load this using pandas, and pass the NumPy arrays to TensorFlow. If you need to scale up to a large set of files, or need a loader that integrates with TensorFlow and tf.data then use the `tf.data.experimental.make_csv_dataset` function:

The only column you need to identify explicitly is the one with the value that the model is intended to predict.

In [5]:
```
 # TODO 1
LABEL_COLUMN = 'survived'
LABELS = [0, 1]
```

Now read the CSV data from the file and create a dataset.

(For the full documentation, see `tf.data.experimental.make_csv_dataset`)

In [6]:
```
# get_dataset() retrieve a Dataverse dataset or its metadata
def get_dataset(file_path, **kwargs):
 # TODO 2
 # Use `tf.data.experimental.make_csv_dataset()` to read CSV files into a dataset.
  dataset = tf.data.experimental.make_csv_dataset(
      file_path,
      batch_size=5, # Artificially small to make examples easier to show.
      label_name=LABEL_COLUMN,
      na_value="?",
      num_epochs=1,
      ignore_errors=True,
      **kwargs)
  return dataset

raw_train_data = get_dataset(train_file_path)
raw_test_data = get_dataset(test_file_path)
```

In [7]:
```
def show_batch(dataset):
  for batch, label in dataset.take(1):
    for key, value in batch.items():
      print("{:20s}: {}".format(key,value.numpy()))
```

Each item in the dataset is a batch, represented as a tuple of (*many examples*, *many labels*). The data from the examples is organized in column-based tensors (rather than row-based tensors), each with as many elements as the batch size (5 in this case).

It might help to see this yourself.

In [8]:
```
show_batch(raw_train_data)
```

```
sex                  : [b'male' b'male' b'male' b'male' b'male']
age                  : [34. 18. 45. 46. 29.]
n_siblings_spouses   : [1 0 1 1 1]
parch                : [0 0 0 0 0]
fare                 : [26.     8.3   83.475 61.175  7.046]
class                : [b'Second' b'Third' b'First' b'First' b'Third']
deck                 : [b'unknown' b'unknown' b'C' b'E' b'unknown']
embark_town          : [b'Southampton' b'Southampton' b'Southampton' b'Southampton'
 b'Southampton']
alone                : [b'n' b'y' b'n' b'n' b'n']
```

As you can see, the columns in the CSV are named. The dataset constructor will pick these names up automatically. If the file you are working with does not contain the column names in the first line, pass them in a list of strings to the `column_names` argument in the `make_csv_dataset` function.

In [9]:
```
CSV_COLUMNS = ['survived', 'sex', 'age', 'n_siblings_spouses', 'parch', 'fare', 'class'

# pass column names in a list of strings to the column_names argument.
temp_dataset = get_dataset(train_file_path, column_names=CSV_COLUMNS)

show_batch(temp_dataset)
```

```
sex                  : [b'male' b'female' b'male' b'male' b'male']
age                  : [30. 50. 18. 51. 28.]
n_siblings_spouses   : [1 0 1 0 0]
parch                : [0 1 1 0 0]
fare                 : [ 16.1   247.521   7.854   8.05    7.05 ]
class                : [b'Third' b'First' b'Third' b'Third' b'Third']
deck                 : [b'unknown' b'B' b'unknown' b'unknown' b'unknown']
embark_town          : [b'Southampton' b'Cherbourg' b'Southampton' b'Southampton' b'South
ampton']
alone                : [b'n' b'n' b'n' b'y' b'y']
```

This example is going to use all the available columns. If you need to omit some columns from the dataset, create a list of just the columns you plan to use, and pass it into the (optional) `select_columns` argument of the constructor.

In [10]:
```
# If you need to omit some columns from the dataset, create a list of just the columns
# pass it into the select_columns argument of the constructor.
SELECT_COLUMNS = ['survived', 'age', 'n_siblings_spouses', 'class', 'deck', 'alone']

temp_dataset = get_dataset(train_file_path, select_columns=SELECT_COLUMNS)

show_batch(temp_dataset)
```

```
age                  : [28. 34. 28. 50.  2.]
n_siblings_spouses   : [0 0 0 2 1]
class                : [b'Third' b'First' b'Third' b'First' b'Second']
```

```
deck                 : [b'unknown' b'unknown' b'unknown' b'unknown' b'unknown']
alone                : [b'y' b'y' b'y' b'n' b'n']
```

Data preprocessing

A CSV file can contain a variety of data types. Typically you want to convert from those mixed types to a fixed length vector before feeding the data into your model.

TensorFlow has a built-in system for describing common input conversions: `tf.feature_column`, see this tutorial for details.

You can preprocess your data using any tool you like (like nltk or sklearn), and just pass the processed output to TensorFlow.

The primary advantage of doing the preprocessing inside your model is that when you export the model it includes the preprocessing. This way you can pass the raw data directly to your model.

Continuous data

If your data is already in an appropriate numeric format, you can pack the data into a vector before passing it off to the model:

In [11]:
```
SELECT_COLUMNS = ['survived', 'age', 'n_siblings_spouses', 'parch', 'fare']
DEFAULTS = [0, 0.0, 0.0, 0.0, 0.0]
temp_dataset = get_dataset(train_file_path,
                           select_columns=SELECT_COLUMNS,
                           column_defaults = DEFAULTS)

show_batch(temp_dataset)
```

```
age                  : [28.   32.5 28.   32.   28. ]
n_siblings_spouses   : [0. 1. 0. 0. 0.]
parch                : [0. 0. 0. 0. 0.]
fare                 : [26.55  30.071  7.829 13.     7.75 ]
```

In [12]:
```
example_batch, labels_batch = next(iter(temp_dataset))
```

Here's a simple function that will pack together all the columns:

In [13]:
```
# `pack()` function will pack together all the columns
def pack(features, label):
# `tf.stack()` stacks a list of rank-R tensors into one rank-(R+1) tensor.
  return tf.stack(list(features.values()), axis=-1), label
```

Apply this to each element of the dataset:

In [14]:
```
packed_dataset = temp_dataset.map(pack)

for features, labels in packed_dataset.take(1):
  print(features.numpy())
  print()
  print(labels.numpy())
```

```
WARNING:tensorflow:AutoGraph could not transform <function pack at 0x7f52c0743ea0> and w
ill run it as-is.
Please report this to the TensorFlow team. When filing the bug, set the verbosity to 10
(on Linux, `export AUTOGRAPH_VERBOSITY=10`) and attach the full output.
Cause: 'arguments' object has no attribute 'posonlyargs'
To silence this warning, decorate the function with @tf.autograph.experimental.do_not_co
nvert
WARNING: AutoGraph could not transform <function pack at 0x7f52c0743ea0> and will run it
as-is.
Please report this to the TensorFlow team. When filing the bug, set the verbosity to 10
(on Linux, `export AUTOGRAPH_VERBOSITY=10`) and attach the full output.
Cause: 'arguments' object has no attribute 'posonlyargs'
To silence this warning, decorate the function with @tf.autograph.experimental.do_not_co
nvert
[[ 18.      1.      0.    108.9  ]
 [ 31.      0.      0.     50.496]
 [ 70.      1.      1.     71.   ]
 [ 24.      1.      0.     16.1  ]
 [ 31.      1.      1.     37.004]]

[0 0 0 0 0]
```

If you have mixed datatypes you may want to separate out these simple-numeric fields. The `tf.feature_column` api can handle them, but this incurs some overhead and should be avoided unless really necessary. Switch back to the mixed dataset:

In [15]:
```
show_batch(raw_train_data)
```

```
sex                 : [b'male' b'female' b'male' b'male' b'male']
age                 : [18. 28. 28. 28. 28.]
n_siblings_spouses  : [0 0 0 3 1]
parch               : [0 0 0 1 1]
fare                : [ 7.75   7.879  7.75  25.467 15.246]
class               : [b'Third' b'Third' b'Third' b'Third' b'Third']
deck                : [b'unknown' b'unknown' b'unknown' b'unknown' b'unknown']
embark_town         : [b'Southampton' b'Queenstown' b'Queenstown' b'Southampton' b'Cherb
ourg']
alone               : [b'y' b'y' b'y' b'n' b'n']
```

In [16]:
```
example_batch, labels_batch = next(iter(temp_dataset))
```

So define a more general preprocessor that selects a list of numeric features and packs them into a single column:

In [17]:
```
class PackNumericFeatures(object):
  def __init__(self, names):
    self.names = names

  def __call__(self, features, labels):
    numeric_features = [features.pop(name) for name in self.names]
    numeric_features = [tf.cast(feat, tf.float32) for feat in numeric_features]
    numeric_features = tf.stack(numeric_features, axis=-1)
    features['numeric'] = numeric_features

    return features, labels
```

In [18]:
```
NUMERIC_FEATURES = ['age','n_siblings_spouses','parch', 'fare']
```

```
packed_train_data = raw_train_data.map(
    PackNumericFeatures(NUMERIC_FEATURES))

packed_test_data = raw_test_data.map(
    PackNumericFeatures(NUMERIC_FEATURES))
```

```
WARNING:tensorflow:AutoGraph could not transform <__main__.PackNumericFeatures object at
0x7f52c06f77b8> and will run it as-is.
Please report this to the TensorFlow team. When filing the bug, set the verbosity to 10
(on Linux, `export AUTOGRAPH_VERBOSITY=10`) and attach the full output.
Cause: module 'gast' has no attribute 'Constant'
To silence this warning, decorate the function with @tf.autograph.experimental.do_not_co
nvert
WARNING: AutoGraph could not transform <__main__.PackNumericFeatures object at 0x7f52c06
f77b8> and will run it as-is.
Please report this to the TensorFlow team. When filing the bug, set the verbosity to 10
(on Linux, `export AUTOGRAPH_VERBOSITY=10`) and attach the full output.
Cause: module 'gast' has no attribute 'Constant'
To silence this warning, decorate the function with @tf.autograph.experimental.do_not_co
nvert
WARNING:tensorflow:AutoGraph could not transform <__main__.PackNumericFeatures object at
0x7f52c06f7438> and will run it as-is.
Please report this to the TensorFlow team. When filing the bug, set the verbosity to 10
(on Linux, `export AUTOGRAPH_VERBOSITY=10`) and attach the full output.
Cause: module 'gast' has no attribute 'Constant'
To silence this warning, decorate the function with @tf.autograph.experimental.do_not_co
nvert
WARNING: AutoGraph could not transform <__main__.PackNumericFeatures object at 0x7f52c06
f7438> and will run it as-is.
Please report this to the TensorFlow team. When filing the bug, set the verbosity to 10
(on Linux, `export AUTOGRAPH_VERBOSITY=10`) and attach the full output.
Cause: module 'gast' has no attribute 'Constant'
To silence this warning, decorate the function with @tf.autograph.experimental.do_not_co
nvert
```

In [19]:
```
show_batch(packed_train_data)
```

```
sex                : [b'male' b'male' b'male' b'female' b'male']
class              : [b'Third' b'Second' b'Third' b'First' b'First']
deck               : [b'unknown' b'unknown' b'unknown' b'B' b'B']
embark_town        : [b'Southampton' b'Southampton' b'Southampton' b'Southampton' b'Che
rbourg']
alone              : [b'n' b'y' b'y' b'n' b'y']
numeric            : [[ 4.     4.     2.    31.275]
 [16.     0.     0.    26.   ]
 [25.     0.     0.     7.05 ]
 [36.     0.     2.    71.   ]
 [32.     0.     0.    30.5  ]]
```

In [20]:
```
example_batch, labels_batch = next(iter(packed_train_data))
```

Data Normalization

Continuous data should always be normalized.

In [21]:
```
# pandas is used for data manipulation and analysis.
import pandas as pd
# pandas module read_csv() function reads the CSV file into a DataFrame object.
desc = pd.read_csv(train_file_path)[NUMERIC_FEATURES].describe()
desc
```

Out[21]:

	age	n_siblings_spouses	parch	fare
count	627.000000	627.000000	627.000000	627.000000
mean	29.631308	0.545455	0.379585	34.385399
std	12.511818	1.151090	0.792999	54.597730
min	0.750000	0.000000	0.000000	0.000000
25%	23.000000	0.000000	0.000000	7.895800
50%	28.000000	0.000000	0.000000	15.045800
75%	35.000000	1.000000	0.000000	31.387500
max	80.000000	8.000000	5.000000	512.329200

In [22]:
```
# TODO 1
MEAN = np.array(desc.T['mean'])
STD = np.array(desc.T['std'])
```

In [23]:
```
def normalize_numeric_data(data, mean, std):
# TODO 2
  # Center the data
  return (data-mean)/std
```

In [24]:
```
print(MEAN, STD)
```

```
[29.631  0.545  0.38  34.385] [12.512  1.151  0.793 54.598]
```

Now create a numeric column. The `tf.feature_columns.numeric_column` API accepts a `normalizer_fn` argument, which will be run on each batch.

Bind the `MEAN` and `STD` to the normalizer fn using `functools.partial`.

In [25]:
```
# See what you just created.
# Bind the MEAN and STD to the normalizer fn using `functools.partial`
normalizer = functools.partial(normalize_numeric_data, mean=MEAN, std=STD)

# `tf.feature_column.numeric_column()` represents real valued or numerical features.
numeric_column = tf.feature_column.numeric_column('numeric', normalizer_fn=normalizer,
numeric_columns = [numeric_column]
numeric_column
```

Out[25]:
```
NumericColumn(key='numeric', shape=(4,), default_value=None, dtype=tf.float32, normalize
r_fn=functools.partial(<function normalize_numeric_data at 0x7f52c066f488>, mean=array
([29.631,  0.545,  0.38 , 34.385]), std=array([12.512,  1.151,  0.793, 54.598])))
```

When you train the model, include this feature column to select and center this block of numeric data:

In [26]:
```
example_batch['numeric']
```

Out[26]:
```
<tf.Tensor: shape=(5, 4), dtype=float32, numpy=
```

```
array([[ 2.   ,  3.   ,  1.   , 21.075],
       [28.   ,  1.   ,  0.   , 16.1  ],
       [28.   ,  1.   ,  0.   , 19.967],
       [16.   ,  0.   ,  1.   , 39.4  ],
       [24.   ,  2.   ,  0.   , 24.15 ]], dtype=float32)>
```

In [27]:
```
# `tf.keras.layers.DenseFeatures()` produces a dense Tensor based on given feature_colu
numeric_layer = tf.keras.layers.DenseFeatures(numeric_columns)
numeric_layer(example_batch).numpy()
```

Out[27]:
```
array([[-2.208,  2.132,  0.782, -0.244],
       [-0.13 ,  0.395, -0.479, -0.335],
       [-0.13 ,  0.395, -0.479, -0.264],
       [-1.089, -0.474,  0.782,  0.092],
       [-0.45 ,  1.264, -0.479, -0.187]], dtype=float32)
```

The mean based normalization used here requires knowing the means of each column ahead of time.

Categorical data

Some of the columns in the CSV data are categorical columns. That is, the content should be one of a limited set of options.

Use the `tf.feature_column` API to create a collection with a `tf.feature_column.indicator_column` for each categorical column.

In [28]:
```
CATEGORIES = {
    'sex': ['male', 'female'],
    'class' : ['First', 'Second', 'Third'],
    'deck' : ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J'],
    'embark_town' : ['Cherbourg', 'Southhampton', 'Queenstown'],
    'alone' : ['y', 'n']
}
```

In [29]:
```
categorical_columns = []
for feature, vocab in CATEGORIES.items():
# Use the `tf.feature_column` API to create a collection with a `tf.feature_column.indi
  cat_col = tf.feature_column.categorical_column_with_vocabulary_list(
        key=feature, vocabulary_list=vocab)
  categorical_columns.append(tf.feature_column.indicator_column(cat_col))
```

In [30]:
```
# See what you just created.
categorical_columns
```

Out[30]:
```
[IndicatorColumn(categorical_column=VocabularyListCategoricalColumn(key='class', vocabul
ary_list=('First', 'Second', 'Third'), dtype=tf.string, default_value=-1, num_oov_bucket
s=0)),
 IndicatorColumn(categorical_column=VocabularyListCategoricalColumn(key='embark_town', v
ocabulary_list=('Cherbourg', 'Southhampton', 'Queenstown'), dtype=tf.string, default_val
ue=-1, num_oov_buckets=0)),
 IndicatorColumn(categorical_column=VocabularyListCategoricalColumn(key='deck', vocabula
ry_list=('A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J'), dtype=tf.string, default_val
ue=-1, num_oov_buckets=0)),
 IndicatorColumn(categorical_column=VocabularyListCategoricalColumn(key='sex', vocabular
```

```
y_list=('male', 'female'), dtype=tf.string, default_value=-1, num_oov_buckets=0)),
 IndicatorColumn(categorical_column=VocabularyListCategoricalColumn(key='alone', vocabul
ary_list=('y', 'n'), dtype=tf.string, default_value=-1, num_oov_buckets=0))]
```

In [31]:

```
# `tf.keras.layers.DenseFeatures()` produces a dense Tensor based on given feature_colu
categorical_layer = tf.keras.layers.DenseFeatures(categorical_columns)
print(categorical_layer(example_batch).numpy()[0])
```

```
[0. 1. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0.]
```

This will be become part of a data processing input later when you build the model.

Combined preprocessing layer

Add the two feature column collections and pass them to a `tf.keras.layers.DenseFeatures` to create an input layer that will extract and preprocess both input types:

In [32]:

```
# Add the two feature column collections
# Pass them to a `tf.keras.layers.DenseFeatures()` to create an input layer.
# TODO 1
preprocessing_layer = tf.keras.layers.DenseFeatures(categorical_columns+numeric_columns
```

In [33]:

```
print(preprocessing_layer(example_batch).numpy()[0])
```

```
[ 0.     1.     0.     0.     1.     0.     0.     0.     0.     0.
  0.     0.     0.     0.     0.     0.     0.     0.    -2.208  2.132
  0.782 -0.244  1.     0.   ]
```

Next Step

A next step would be to build a build a `tf.keras.Sequential`, starting with the `preprocessing_layer`, which is beyond the scope of this lab. We will cover the Keras Sequential API in the next Lesson.

Load NumPy data

Load necessary libraries

First, restart the Kernel. Then, we will start by importing the necessary libraries for this lab.

In [1]:

```
# Importing the necessary libraries
import numpy as np
import tensorflow as tf

print("TensorFlow version: ",tf.version.VERSION)
```

```
TensorFlow version:  2.3.0-dev20200613
```

Load data from `.npz` file

We use the MNIST dataset in Keras.

In [3]:
```
DATA_URL = 'https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz'

# `tf.keras.utils.get_file()` downloads a file from a URL if it not already in the cach
path = tf.keras.utils.get_file('mnist.npz', DATA_URL)
with np.load(path) as data:
# TODO 1
  train_examples = data['x_train']
  train_labels = data['y_train']
  test_examples = data['x_test']
  test_labels = data['y_test']
```

Load NumPy arrays with `tf.data.Dataset`

Assuming you have an array of examples and a corresponding array of labels, pass the two arrays as a tuple into `tf.data.Dataset.from_tensor_slices` to create a `tf.data.Dataset` .

In [4]:
```
# With the help of `tf.data.Dataset.from_tensor_slices()` method, we can get the slices
# by using `tf.data.Dataset.from_tensor_slices()` method.
# TODO 2
train_dataset = tf.data.Dataset.from_tensor_slices((train_examples, train_labels))
test_dataset = tf.data.Dataset.from_tensor_slices((test_examples, test_labels))
```

Next Step

A next step would be to build a build a `tf.keras.Sequential` , starting with the `preprocessing_layer` , which is beyond the scope of this lab. We will cover the Keras Sequential API in the next Lesson.

Resources

1. Load text data - this link: https://www.tensorflow.org/tutorials/load_data/text
2. TF.text - this link: https://www.tensorflow.org/tutorials/tensorflow_text/intro
3. Load image daeta - https://www.tensorflow.org/tutorials/load_data/images
4. Read data into a Pandas DataFrame - https://www.tensorflow.org/tutorials/load_data/pandas_dataframe
5. How to represent Unicode strings in TensorFlow - https://www.tensorflow.org/tutorials/load_data/unicode
6. TFRecord and tf.Example - https://www.tensorflow.org/tutorials/load_data/tfrecord