Input to a Deep Neural Network

tf.data makes input pipelines in TensorFlow to be

- Fast
- Flexible
- Easy-to-use

Neural networks expect numbers as inputs

Real-life input data may contain non-numerical data such as strings, numbers that represent categories. We can create input pipelines to process these inputs into the form that can be used by the neural network.

Example image model pipeline

- Aggregate data from files in a distributed file system
- Apply random perturbations to each image
- Merge randomly selected images into a batch for training

Example text model pipeline

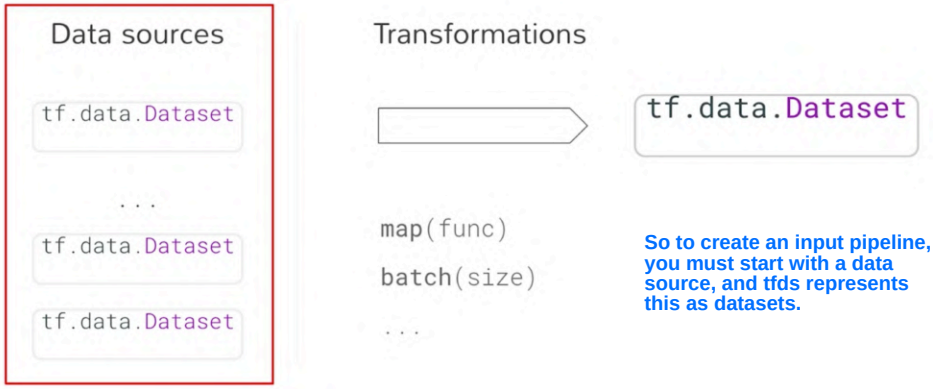
- Extracting symbols from raw text data
- Converting them to embedding identifiers with a lookup table
- Batching together sequences of different lengths

Basic mechanics

Often you don't want to use the raw data, but some form of transformation on it. So for example, with image processing, you may want to apply augmentation techniques to prevent over fitting.

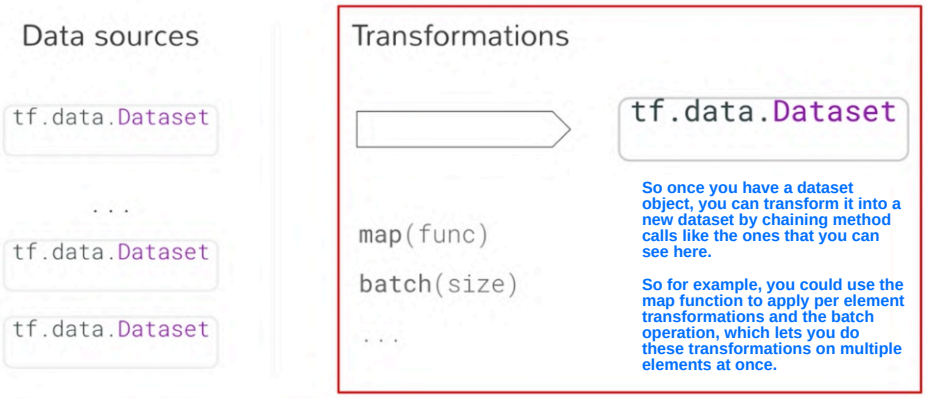
With this in mind, the tf.data API introduces a tf.data.Dataset abstraction that represents a sequence of elements in which each item consists of one or more components, which might be the image and it's label for example

Basic mechanics



So to create an input pipeline, you must start with a data source, and tfds represents this as datasets.

Basic mechanics



Using an iterator to navigate

```
dataset = tf.data.Dataset.from_tensor_slices([1, 2, 3, 4])
it = iter(dataset)

>>> while True:
      try:
        print(next(it))
      except StopIteration as e:
        break
tf.Tensor(1, shape=(), dtype=int32)
tf.Tensor(2, shape=(), dtype=int32)
tf.Tensor(3, shape=(), dtype=int32)
tf.Tensor(4, shape=(), dtype=int32)
```

For example, you could create a very simple dataset using in-memory data like this, simply passing a list to from tensor slices.

Once I have the dataset, I can then create an iterator to go through it.

Loading numpy arrays (from_tensor_slices)

```
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.cifar10.load_data()

dataset = tf.data.Dataset.from_tensor_slices((x_train, y_train))

>>> for image, label in tfds.as_numpy(dataset.take(2)):
      print(image.shape, label)
(32, 32, 3) [6]
(32, 32, 3) [9]
```

The same technique can be used with real data. So here, we use the cifar10 dataset from keras, and use tf.data.dataset.fromtensorslices to turn this into a dataset. We can now iterate through this dataset or a subset of it like shown here. Then we could do something like using NumPy to print out the dimensions of the images and their labels.

First	Last	Addr	Phone	Gender	Age	
Jane	Smith	123 Anywhere	555 555 5555	1	3	

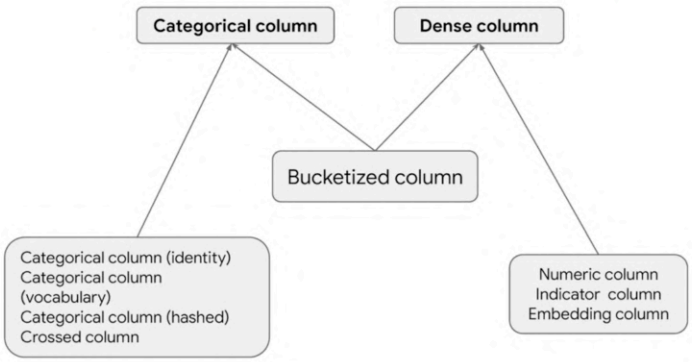
Index	Description
0	Male
1	Female
2	Nonbinary
3	Trans
4	Unassigned
...	...

First	Last	Addr	Phone	Gender	Age	
Jane	Smith	123 Anywhere	555 555 5555	1	3	

Index	Description
0	Male
1	Female
2	Nonbinary
3	Trans
4	Unassigned
...	...

Index	Description
0	Infant
1	Child
2	Teen
3	Young Adult
4	Adult
...	...

Primer on Feature Columns



Here are all the different types of feature columns available in TensorFlow which can be used by tf.data seamlessly with no issues whatsoever.

At a higher level, you see that there are categorical columns and dense columns for numeric and embedded features.

However, there is an exception with the bucketized-column inheriting from both categorical and dense columns.

https://archive.ics.uci.edu/ml/datasets/iris

sepal_length	sepal_width	petal_length	petal_width	species
5.1	3.5	1.4	0.2	Iris-setosa
4.9	3	1.4	0.2	Iris-setosa
4.7	3.2	1.3	0.2	Iris-setosa
4.6	3.1	1.5	0.2	Iris-setosa
5	3.6	1.4	0.2	Iris-setosa
5.4	3.9	1.7	0.4	Iris-setosa
4.6	3.4	1.4	0.3	Iris-setosa
5	3.4	1.5	0.2	Iris-setosa
4.4	2.9	1.4	0.2	Iris-setosa
4.9	3.1	1.5	0.1	Iris-setosa
5.4	3.7	1.5	0.2	Iris-setosa
4.8	3.4	1.6	0.2	Iris-setosa
4.8	3	1.4	0.1	Iris-setosa
4.3	3	1.1	0.1	Iris-setosa
5.8	4	1.2	0.2	Iris-setosa
5.7	4.4	1.5	0.4	Iris-setosa

Numeric column

The Iris dataset has all numeric data as its input features:

- SepalLength
- SepalWidth
- PetalLength
- PetalWidth

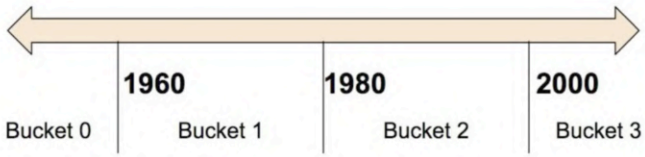
Specifying data types

```
# Defaults to a tf.float32 scalar.
numeric_feature_column = tf.feature_column.numeric_column(key="SepalLength")

# Represent a tf.float64 scalar.
numeric_feature_column = tf.feature_column.numeric_column(key="SepalLength",
                                                          dtype=tf.float64)
```

Shapes for different numeric data

```
# Represent a 10-element vector in which each cell contains a tf.float32.
vector_feature_column = tf.feature_column.numeric_column(key="Bowling",
                                                         shape=10)

# Represent a 10x5 matrix in which each cell contains a tf.float32.
matrix_feature_column = tf.feature_column.numeric_column(key="MyMatrix",
                                                         shape=[10,5])
```

Bucketized column

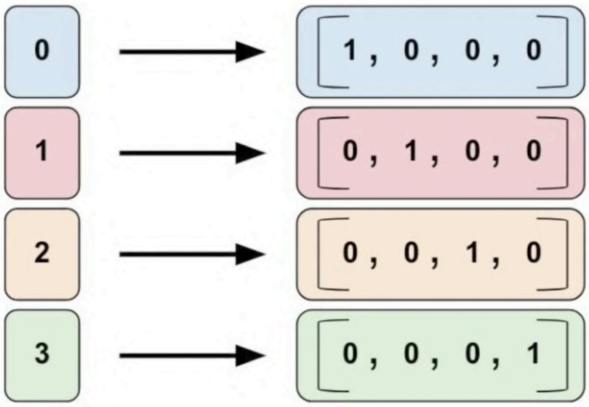


Date Range	Represented as...
< 1960	[1, 0, 0, 0]
>= 1960 but < 1980	[0, 1, 0, 0]
>= 1980 but < 2000	[0, 0, 1, 0]
>= 2000	[0, 0, 0, 1]

Bucketizing features

```
# First, convert the raw input to a numeric column.
numeric_feature_column = tf.feature_column.numeric_column("Year")

# Then, bucketize the numeric column on the years 1960, 1980, and 2000.
bucketized_feature_column = tf.feature_column.bucketized_column(
    source_column = numeric_feature_column,
    boundaries = [1960, 1980, 2000])
```

Categorical identity column



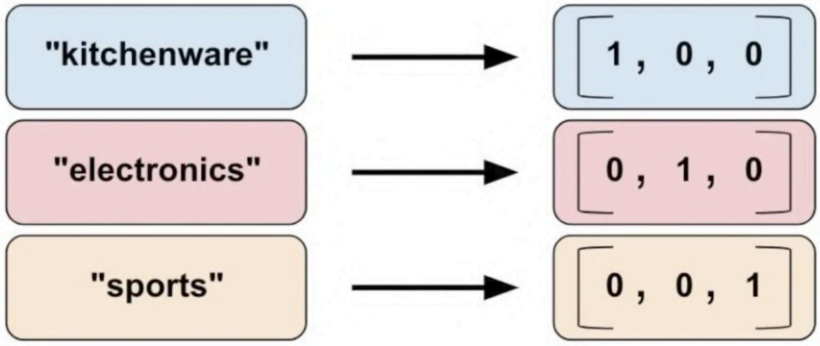
A special case of bucketize columns can be seen with categorical identity columns, where instead of ranges, we can have straight values that are one-hot encoded. So 0 might be 1, 0, 0 and 1 might be 0, 1, 0, etc.

Categorizing identity features

```
identity_feature_column = tf.feature_column.categorical_column_with_identity(
    key='my_feature_b',
    num_buckets=4) # Values [0, 4]


def input_fn():
    ...
    return ({ 'my_feature_a':[7, 9, 5, 2], 'my_feature_b':[3, 1, 2, 2] },
            [Label_values])
```

- Here is the example code that calls tf.feature_column.categorical_column_with_identity to implement a categorical identity column.
- Create categorical output for an integer feature named "my_feature_b".
- The values of my_feature_b must be greater than or equal to zero and less than the number of buckets.
- For the preceding call to work, the input_fn() must return a dictionary containing 'my_feature_b' as a key.
- The values assigned to 'my_feature_b' must belong to the set of numbers from 0 to 4, excluding 4.

Categorical vocabulary column



A great way to handle strings if we have a fixed range of keywords were working from is to use a categorical vocabulary column. So words like kitchenware, electronics and sports can be one-hot encoded to give arrays like these.

Creating a categorical vocab column

To create one from a list, you specify the list with a vocabulary list parameter and then call categorical column with vocabulary list.

From a vocabulary list

```
vocabulary_feature_column = tf.feature_column.categorical_column_with_vocabulary_list(
                            key=feature_name,
                            vocabulary_list=["kitchenware", "electronics", "sports"])
```

From a vocabulary file

```
vocabulary_feature_column = tf.feature_column.categorical_column_with_vocabulary_file(
                            key=feature_name,
                            vocabulary_file="product_class.txt",
                            vocabulary_size=3)
```

Creating a categorical vocab column

Or if your words are in a file, you just point the vocabulary file parameter at the file, then specify the desired vocab size.

From a vocabulary list

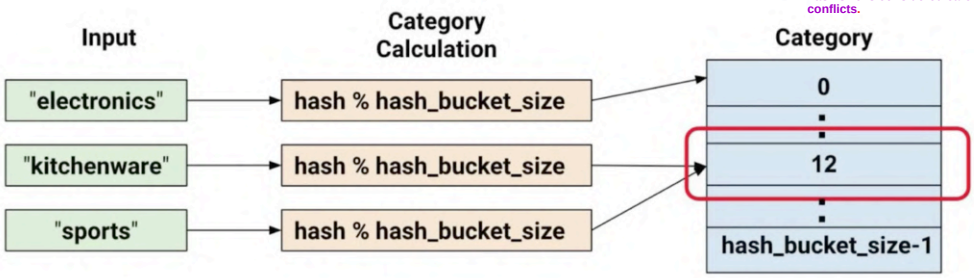
```
vocabulary_feature_column = tf.feature_column.categorical_column_with_vocabulary_list(
                            key=feature_name,
                            vocabulary_list=["kitchenware", "electronics", "sports"])
```

From a vocabulary file

```
vocabulary_feature_column = tf.feature_column.categorical_column_with_vocabulary_file(
                            key=feature_name,
                            vocabulary_file="product_class.txt",
                            vocabulary_size=3)
```

Hashed column

```
hash(raw_feature) % hash_bucket_size
```



This works nicely if there's a small number of categories, and the examples we showed earlier had only three or four.

A common technique used when there are many categories is to hash them to a smaller set.

Of course you have to be a little careful with this approach where unrelated inputs might be mapped into the same category and consequently mean the same thing to the neural network. You can see that type of scenario here where kitchenware in sports both get assigned to a category based on the bucket they end up in after hashing. This is a common problem with hashing algorithms. So make sure your hash size is considered carefully to avoid too many conflicts.

Hashed column

```
hashed_feature_column = tf.feature_column.categorical_column_with_hash_bucket(
                        key="some_feature",
                        hash_bucket_size=100) # The number of categories
```

Crossed column

```
# Bucketize the latitude and longitude using the `edges`
latitude_bucket_fc = tf.feature_column.bucketized_column(
    tf.feature_column.numeric_column('latitude'),
    list(atlanta.latitude.edges))

longitude_bucket_fc = tf.feature_column.bucketized_column(
    tf.feature_column.numeric_column('longitude'),
    list(atlanta.longitude.edges))

# Cross the bucketized columns, using 5000 hash bins.
crossed_lat_lon_fc = tf.feature_column.crossed_column(
    [latitude_bucket_fc, longitude_bucket_fc], 5000)
```

To do this in code, you use the feature column, categorical column with hash bucket method. It looks something like this. This hashed column can be used to represent data with hash buckets. As with many counter-intuitive phenomena and machine-learning, it turns out that hashing often works very well in practice.

That's because hash categories provide the model with some separation. The model can use additional features to make the distinction more apparent. Combining features into a single feature, also known as feature crosses, enables a model to learn separate weights for each combination of features. For this, we'll consider a dataset which hosts features such as latitudes and longitudes. Here you can see that those features have been bucketised.

Feature crosses are synthetic featured formed by multiplying (or crossing) two or more features. Crossing features can provide predictive abilities beyond what those features could provide individually.

To learn more about feature crosses, check out this video from Google's Machine Learning Crash Course.

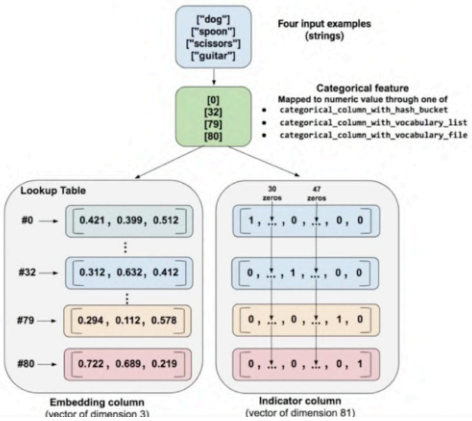
https://developers.google.com/machine-learning/crash-course/feature-crosses/video-lecture

Crossed column

```
# Bucketize the latitude and longitude using the `edges`
latitude_bucket_fc = tf.feature_column.bucketized_column(
    tf.feature_column.numeric_column('latitude'),
    list(atlanta.latitude.edges))

longitude_bucket_fc = tf.feature_column.bucketized_column(
    tf.feature_column.numeric_column('longitude'),
    list(atlanta.longitude.edges))

# Cross the bucketized columns, using 5000 hash bins.
crossed_lat_lon_fc = tf.feature_column.crossed_column(
    [latitude_bucket_fc, longitude_bucket_fc], 5000)
```

Now let's create a cross column by creating a new feature that takes both latitude and longitude into the feature sets. We'll make a tf.feature_column.crossed_column function and pass the list of the features to it. Note that crossed column does not build the full table of all possible combinations which could be very large. Instead, it is backed by a hashed column, so that you can choose how large the table is by specifying the hash bucket size, which is 5,000 in this example.

Embedding column



One downside with one hot encoded vectors that you've seen so far is that they can lead to some very large sparse and inefficient vectors embedding columns work very well to avoid this.

Now. Let's look at an example comparing indicator and embedding columns. Suppose our input values consists of different words with a vocabulary size of 81. When an example is processed one of the categorical feature column functions Maps the example string to a numerical categorical value. So for example a function might map spoon to 32. The actual values of the embedding vectors depend on the mapping function.

When using an indicator column a function converts each numeric categorical value into an 81 element Vector. This is because our palette consisted of 81 words. However, an embedding column handles this much better by using a function that maps the numerical categorical values to indices in a lookup table each. Lot in that lookup table contains a three element Vector.

So how do the values in the embedding vectors magically get assigned? These assignments happen during the training and that is the model learns the best way to map your input numeric categorical values to the embeddings vector value to solve your problem.

Embedding columns increase your models capabilities since an embeddings vector learns new relationships between categories from the training data.

Embedding column

```
embedding_dimensions =  number_of_categories**0.25
categorical_column = ... # Create any categorical column

# Represent the categorical column as an embedding column.
# This means creating an embedding vector lookup table with one element for each
category.
embedding_column = tf.feature_column.embedding_column(
    categorical_column=categorical_column,
    dimension=embedding_dimensions)
```

So for a quick look at how an embedded column works, here's the code. Does the rule of thumb that the number of embedding dimensions should be in the same order as the fourth root of the number of categories? So in this case, we have 81 categories, so three dimensions actually works really well.

We can calculate the number of dimensions like this by setting the number of dimensions to the 4th root of the number of categories.

Embedding column

```
embedding_dimensions =  number_of_categories**0.25
categorical_column = ... # Create any categorical column

# Represent the categorical column as an embedding column.
# This means creating an embedding vector lookup table with one element for each
category.
embedding_column = tf.feature_column.embedding_column(
    categorical_column=categorical_column,
    dimension=embedding_dimensions)
```

Then we can make our column to be an embedding column built from the categorical one using the number of embeddings that we just calculated.

Data sources

Numpy	DataFrames	Images
CSV and Text	TFRecords	Generators

Numpy — Loading a dataset from npz

```
# Download dataset
DATA_URL = 'https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz'
path = tf.keras.utils.get_file('mnist.npz', DATA_URL)

# Extract train and test examples
with np.load(path) as data:
  train_examples = data['x_train']
  train_labels = data['y_train']
  test_examples = data['x_test']

# Create train and test datasets out of the examples
train_dataset = tf.data.Dataset.from_tensor_slices((train_examples, train_labels))
test_dataset = tf.data.Dataset.from_tensor_slices(test_examples)

for feat, targ in train_dataset.take(2):
  print ('Features shape: {}, Target: {}'.format(feat.shape, targ))
Features shape: (28, 28), Target: 5
Features shape: (28, 28), Target: 0
```

Let's start with the most basic and widely available data source, which is NumPy. We'll begin by downloading the mnist data in NumPy's NPZ format.

You're most likely to use a numpy array if all of your input data fits into memory. Otherwise, working with a large NumPy dataset is usually not recommended.

The simplest way to read a NumPy array, is by employing the tf.data.dataset from Tensor slices function. For the training datasets, we're loading both the train examples and labels as a tuple with tf.data.

One thing to keep in mind is that, while you're passing multiple elements inside the tuple into the from Tensor slices function, make sure the number of items in each of the components is the same.

Now, we can iterate through our dataset, taking a couple of records and printing their shape and the targets, and then we can see that the first record is a 28 by 28, and the target or label is five. Similarly, the next one is the standard mnist 28 by 28, but it's target label is zero.

Pandas — Create DataFrames out of CSVs

```
csv_file = tf.keras.utils.get_file('heart.csv', 'https://storage.googleapis.com/applied-dl/heart.csv')
df = pd.read_csv(csv_file)
df.head()
```

	age	sex	cp	trestbps	chol	fbs	restecg	thalach	exang	oldpeak	slope	ca	thal	target
0	63	1	1	145	233	1	2	150	0	2.3	3	0	fixed	0
1	67	1	4	160	286	0	2	108	1	1.5	2	3	normal	1
2	67	1	4	120	229	0	2	129	1	2.6	2	2	reversible	0
3	37	1	3	130	250	0	0	187	0	3.5	3	0	normal	0
4	41	0	2	130	204	0	2	172	0	1.4	1	0	normal	0

Pandas — Discretizing features

```
df['thal'] = pd.Categorical(df['thal'])
df['thal'] = df.thal.cat.codes
df.head()
```

	age	sex	cp	trestbps	chol	fbs	restecg	thalach	exang	oldpeak	slope	ca	thal	target
0	63	1	1	145	233	1	2	150	0	2.3	3	0	**2**	0
1	67	1	4	160	286	0	2	108	1	1.5	2	3	**3**	1
2	67	1	4	120	229	0	2	129	1	2.6	2	2	**4**	0
3	37	1	3	130	250	0	0	187	0	3.5	3	0	**3**	0
4	41	0	2	130	204	0	2	172	0	1.4	1	0	**3**	0

Pandas — Dataset from features and targets

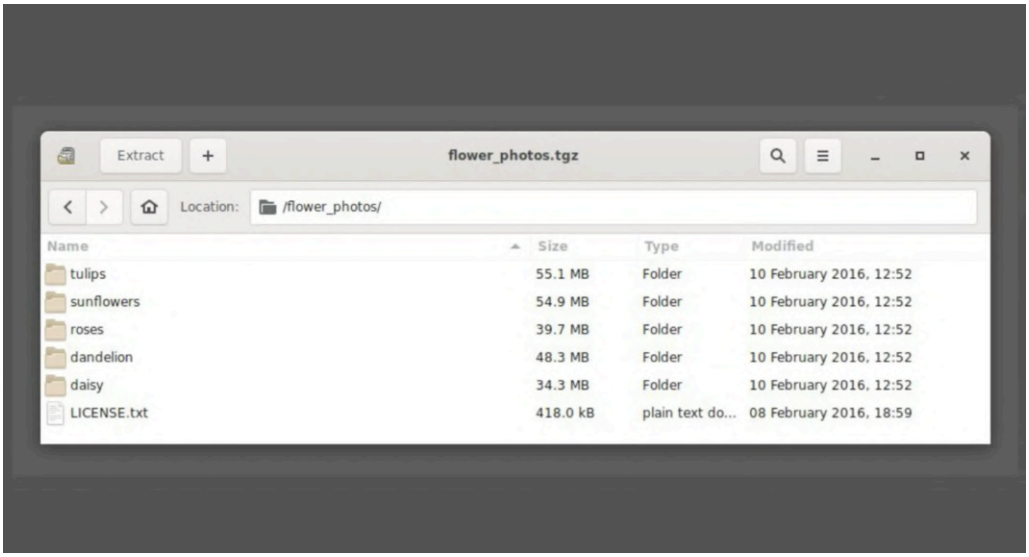
```
target = df.pop('target')
dataset = tf.data.Dataset.from_tensor_slices((df.values, target.values))

>>> for feat, targ in dataset.take(5):
      print ('Features: {}, Target: {}'.format(feat, targ))

Features: [ 63.   1.   1. 145. 233.   1.   2. 150.   0.   2.3  3.   0.   2. ], Target: 0
Features: [ 67.   1.   4. 160. 286.   0.   2. 108.   1.   1.5  2.   3.   3. ], Target: 1
Features: [ 67.   1.   4. 120. 229.   0.   2. 129.   1.   2.6  2.   2.   4. ], Target: 0
Features: [ 37.   1.   3. 130. 250.   0.   0. 187.   0.   3.5  3.   0.   3. ], Target: 0
Features: [ 41.   0.   2. 130. 204.   0.   2. 172.   0.   1.4  1.   0.   3. ], Target: 0
```

Images — Download and extract images

Another type of data that's commonly seen is Images. Often in an archive like a zip file, labeling this can be tricky. It's often done based on the filename or perhaps the subdirectory containing the images.

Sometimes you don't have that luxury, and you have to have a lookup file, with the filename and the label. This code is a simple example of where you can download an archive containing images and named subdirectories, and then use Keras utilities to load and untar them.

```
import pathlib

DATA_URL =
'https://storage.googleapis.com/download.tensorflow.org/example_images/flower_photos.tgz'
data_root_orig = tf.keras.utils.get_file(origin=DATA_URL,
                                         fname='flower_photos', untar=True)
data_root = pathlib.Path(data_root_orig)

label_names = sorted(item.name for item in data_root.glob('*/') if item.is_dir())
>>> label_names
['daisy', 'dandelion', 'roses', 'sunflowers', 'tulips']
```



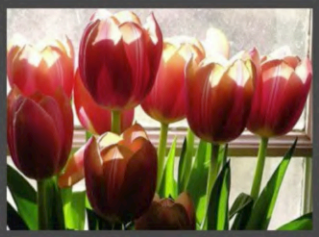
Images — Download and extract images

So now, if I go through the subdirectories, I can get the label names, and I can read the images from those subdirectories and assign them to the specific label. Given that I have the images in subdirectories after on archiving, I can now go through and for example, just pick a random image in a random subdirectory and render it.

```
import pathlib

DATA_URL =
'https://storage.googleapis.com/download.tensorflow.org/example_images/flower_photos.tgz'
data_root_orig = tf.keras.utils.get_file(origin=DATA_URL,
                                         fname='flower_photos', untar=True)
data_root = pathlib.Path(data_root_orig)

label_names = sorted(item.name for item in data_root.glob('*/') if item.is_dir())
>>> label_names
['daisy', 'dandelion', 'roses', 'sunflowers', 'tulips']
```

Images — Display a random sample from the loaded dataset

```
import random
import IPython.display as display

all_image_paths = list(data_root.glob('*/*'))
all_image_paths = [str(path) for path in all_image_paths]
random.shuffle(all_image_paths)

image_count = len(all_image_paths)
image_count

image_path = random.choice(all_image_paths)
display.display(display.Image(image_path))
```



CSV

Loading the structured dataset

```
TRAIN_DATA_URL = "https://storage.googleapis.com/tf-datasets/titanic/train.csv"
train_file_path = tf.keras.utils.get_file("train.csv", TRAIN_DATA_URL)

df = pd.read_csv(train_file_path, sep=',')
df.head()
```

What is overwhelming in this dataset is the amount of diversity in the kinds of data that you get to deal with. Hence, making use of feature columns can undoubtedly be beneficial in this case. There's lots of ways to read a CSV, but I like using the keras-utilities to specify the file path, and then pandas read_csv to read it and get it into a DataFrame.

	survived	sex	age	n_siblings_spouses	parch	fare	class	deck	embark_town	alone
0	0	male	22.0	1	0	7.2500	Third	unknown	Southampton	n
1	1	female	38.0	1	0	71.2833	First	C	Cherbourg	n
2	1	female	26.0	0	0	7.9250	Third	unknown	Southampton	y
3	1	female	35.0	1	0	53.1000	First	C	Southampton	n
4	0	male	28.0	0	0	8.4583	Third	unknown	Queenstown	y

CSV

Numeric data

```
NUMERIC_FEATURES = ['age','n_siblings_spouses','parch', 'fare']
dense_df = df[NUMERIC_FEATURES]
dense_df.head()
```

	age	n_siblings_spouses	parch	fare
0	22.0	1	0	7.2500
1	38.0	1	0	71.2833
2	26.0	0	0	7.9250
3	35.0	1	0	53.1000

CSV

Leveraging features columns

```
numeric_columns = []
for feature in NUMERIC_FEATURES:
  num_col = tf.feature_column.numeric_column(feature)
  numeric_columns.append(tf.feature_column.indicator_column(num_col))
```

If I want to use feature columns, I could do something like this. Again, I've gone through my set of desired features from the previous slide, the name of the column.

```
>>> numeric_columns
[IndicatorColumn(categorical_column=NumericColumn(key='age', shape=(1,),
default_value=None, dtype=tf.float32, normalizer_fn=None)),
 IndicatorColumn(categorical_column=NumericColumn(key='n_siblings_spouses', shape=(1,),
default_value=None, dtype=tf.float32, normalizer_fn=None)),
 IndicatorColumn(categorical_column=NumericColumn(key='parch', shape=(1,),
default_value=None, dtype=tf.float32, normalizer_fn=None)),
 IndicatorColumn(categorical_column=NumericColumn(key='fare', shape=(1,),
default_value=None, dtype=tf.float32, normalizer_fn=None))]
```

CSV

Leveraging features columns

```
numeric_columns = []
for feature in NUMERIC_FEATURES:
  num_col = tf.feature_column.numeric_column(feature)
  numeric_columns.append(tf.feature_column.indicator_column(num_col))

>>> numeric_columns
[IndicatorColumn(categorical_column=NumericColumn(key='age', shape=(1,),
default_value=None, dtype=tf.float32, normalizer_fn=None)),
 IndicatorColumn(categorical_column=NumericColumn(key='n_siblings_spouses', shape=(1,),
default_value=None, dtype=tf.float32, normalizer_fn=None)),
 IndicatorColumn(categorical_column=NumericColumn(key='parch', shape=(1,),
default_value=None, dtype=tf.float32, normalizer_fn=None)),
 IndicatorColumn(categorical_column=NumericColumn(key='fare', shape=(1,),
default_value=None, dtype=tf.float32, normalizer_fn=None))]
```

Then for each of these, I can specify the num_col as a numeric column that I'll create from that feature

CSV

Leveraging features columns

```
numeric_columns = []
for feature in NUMERIC_FEATURES:
  num_col = tf.feature_column.numeric_column(feature)
  numeric_columns.append(tf.feature_column.indicator_column(num_col))

>>> numeric_columns
[IndicatorColumn(categorical_column=NumericColumn(key='age', shape=(1,),
default_value=None, dtype=tf.float32, normalizer_fn=None)),
 IndicatorColumn(categorical_column=NumericColumn(key='n_siblings_spouses', shape=(1,),
default_value=None, dtype=tf.float32, normalizer_fn=None)),
 IndicatorColumn(categorical_column=NumericColumn(key='parch', shape=(1,),
default_value=None, dtype=tf.float32, normalizer_fn=None)),
 IndicatorColumn(categorical_column=NumericColumn(key='fare', shape=(1,),
default_value=None, dtype=tf.float32, normalizer_fn=None))]
```

But more importantly, I can create a collection of numeric columns that I'll add an indicator column for that column

CSV

Leveraging features columns

```
numeric_columns = []
for feature in NUMERIC_FEATURES:
  num_col = tf.feature_column.numeric_column(feature)
  numeric_columns.append(tf.feature_column.indicator_column(num_col))

>>> numeric_columns
[IndicatorColumn(categorical_column=NumericColumn(key='age', shape=(1,),
default_value=None, dtype=tf.float32, normalizer_fn=None)),
 IndicatorColumn(categorical_column=NumericColumn(key='n_siblings_spouses', shape=(1,),
default_value=None, dtype=tf.float32, normalizer_fn=None)),
 IndicatorColumn(categorical_column=NumericColumn(key='parch', shape=(1,),
default_value=None, dtype=tf.float32, normalizer_fn=None)),
 IndicatorColumn(categorical_column=NumericColumn(key='fare', shape=(1,),
default_value=None, dtype=tf.float32, normalizer_fn=None))]
```

So now, when I output my collection of numeric columns, you can see that they're all Indicator columns.

CSV

Leveraging features columns

Ffor age, we now have it as an indicator column that contains a float 32. It's not normalized and its shape is shown.

```
>>> numeric_columns
[IndicatorColumn(categorical_column=NumericColumn(key='age',
shape=(1,), default_value=None, dtype=tf.float32,
normalizer_fn=None)),
...
```

CSV

Categorical data

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

cat_df = df[list(CATEGORIES.keys())]
cat_df.head()
```

If the data is categorical, you can also specify how it can be formatted. So for example, if you recall the gender fields could contain male or female, lets now turn them into explicit categories. This data was collected back in 1912 when the Titanic sailed, and in that case, there were only options for expressing gender as male and female, and the dataset only contains these values. Similarly, we could express that will only take three classes of travel for a second and third, and formalize those as the class.

CSV

Categorical data

```
cat_df = df[list(CATEGORIES.keys())]
cat_df.head()
```

The results of this will be this dataset, just containing the categorical sets.

```
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']
}
```

	sex	class	deck	embark_town	alone
0	male	Third	unknown	Southampton	n
1	female	First	C	Cherbourg	n
2	female	Third	unknown	Southampton	y
3	female	First	C	Southampton	n
4	male	Third	unknown	Queenstown	y

CSV

Categorical columns from raw data

Let's iterate through the categories that we've just specified. Feature is the name of the feature, and vocab is the list for it.

So male and female for sex, for a second and third for class, etc.

```
categorical_columns = []
for feature, vocab in CATEGORIES.items():
  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))
```

CSV

Categorical columns from raw data

Now, we can create a category column from a vocabulary list. The key for this is the feature, and the vocabulary list is the collection of vocab for that feature. Again, this is male and female for sex, for a second and third for class.

```
categorical_columns = []
for feature, vocab in CATEGORIES.items():
  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))
```

CSV

Categorical columns from raw data

Then I'll just append them to a list of categorical columns to keep a collection of them.

```
categorical_columns = []
for feature, vocab in CATEGORIES.items():
  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))
```

CSV

Categorical columns from raw data

```
>>> categorical_columns
[IndicatorColumn(categorical_column=VocabularyListCategoricalColumn(key='sex', vocabulary_list=('male', 'female'), dtype=tf.string, default_value=-1, num_oov_buckets=0)),

 IndicatorColumn(categorical_column=VocabularyListCategoricalColumn(key='class', vocabulary_list=('First', 'Second', 'Third'), dtype=tf.string, default_value=-1, num_oov_buckets=0)),
 ...
```

Similarly, the class of transport is an indicator column with a vocabulary lists it is strings and there's nothing out of vocab.

Text

Loading texts with TextLineDataset

```
DIRECTORY_URL =
'https://storage.googleapis.com/download.tensorflow.org/data/illiad/'
FILE_NAME = 'cowper.txt'
```

cowper.txt

```
1  Achilles sing, O Goddess! Peleus' son;
2  His wrath pernicious, who ten thousand woes
3  Caused to Achaia's host, sent many a soul
4  Illustrious into Ades premature,
5  And Heroes gave (so stood the will of Jove)
6  To dogs and to all ravening fowls a prey,
7  When fierce dispute had separated once
8  The noble Chief Achilles from the son
9  Of Atreus, Agamemnon, King of men.
10 Who them to strife impell'd? What power divine?
11 Latona's son and Jove's. For he, incensed
12 Against the King, a foul contagion raised
```

Text — Loading texts with TextLineDataset

When you download it, you'll see that it's poetry that's on separate lines like this. I can now use a text line dataset pointed at the file path to get the dataset as a series of text lines.

```
file_path = tf.keras.utils.get_file(name,
                origin=DIRECTORY_URL + FILE_NAME)

lines_dataset = tf.data.TextLineDataset(file_path)
```

Text — Inspecting texts

```
>>> for text_data in tfds.as_numpy(lines_dataset.take(3)):
      print(text_data.decode('utf-8'))


Achilles sing, O Goddess! Peleus' son;
His wrath pernicious, who ten thousand woes
Caused to Achaia's host, sent many a soul
```

TFRecord — Reading TFRecord files

```
filenames = [tf_record_filename]
raw_dataset = tf.data.TFRecordDataset(filenames)
```

```
feature_description = {
        'feature1': tf.io.FixedLenFeature((), tf.string),
        'feature2': tf.io.FixedLenFeature((), tf.int64)
}

for raw_record in raw_dataset.take(1):
  example = tf.io.parse_single_example(raw_record, feature_description)
  print(example)
```

What if you have too much data to having a single file or archive and that you might have to stream it all over network. Well, in this case, TF Record is your friend.

To use TF Record, you might have a lot of filenames of different records. So you'll initialize your dataset from them by passing them to tf.data.TFrecordDataset.

TFRecord — Reading TFRecord files

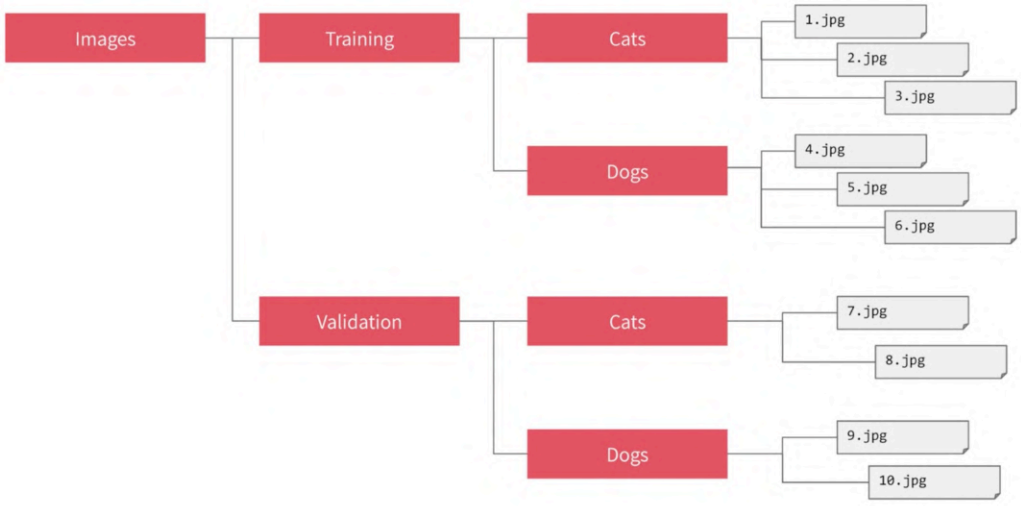
```
filenames = [tf_record_filename]
raw_dataset = tf.data.TFRecordDataset(filenames)

feature_description = {
        'feature1': tf.io.FixedLenFeature((), tf.string),
        'feature2': tf.io.FixedLenFeature((), tf.int64)
}

for raw_record in raw_dataset.take(1):
  example = tf.io.parse_single_example(raw_record, feature_description)
  print(example)
```

You then specify the features that you want to use. So here as an example, we're saying that we're interested in two features. The first feature 1 is a string, and the second feature 2 is an int64. The fixed len feature indicates that in the dataset, the feature has a fixed size. The number of elements in the string, for example, could be static.

TFRecord — Reading TFRecord files

```
filenames = [tf_record_filename]
raw_dataset = tf.data.TFRecordDataset(filenames)

feature_description = {
        'feature1': tf.io.FixedLenFeature((), tf.string),
        'feature2': tf.io.FixedLenFeature((), tf.int64)
}

for raw_record in raw_dataset.take(1):
  example = tf.io.parse_single_example(raw_record, feature_description)
  print(example)
```

As before, I can now go through my dataset and parse it out to see what's in it.



Generators — Keras ImageDataGenerator

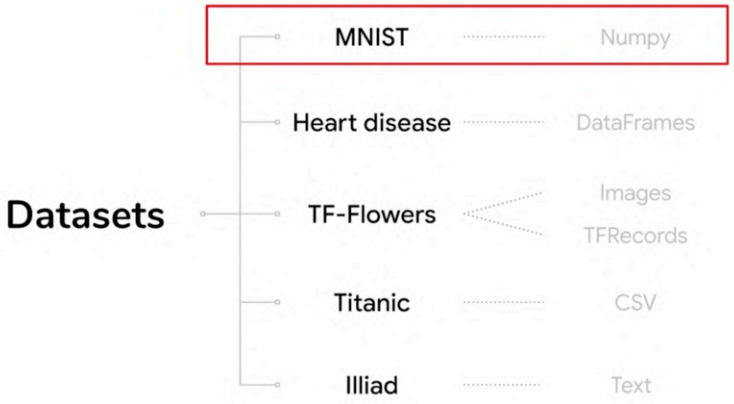
```
def make_generator():
  train_datagen = tf.keras.preprocessing.image.ImageDataGenerator(rescale=1. / 255,
                    rotation_range=20, zoom_range=[0.8, 1.2])

  train_generator = train_datagen.flow_from_directory(catsdogs,
                    target_size=(224, 224), class_mode='categorical',batch_size=32)

  return train_generator


train_generator = tf.data.Dataset.from_generator(
                          make_generator,(tf.float32, tf.uint8))
```

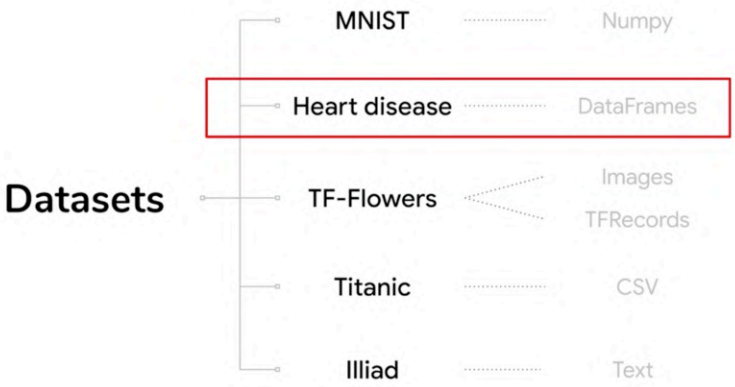
Generators — Keras ImageDataGenerator

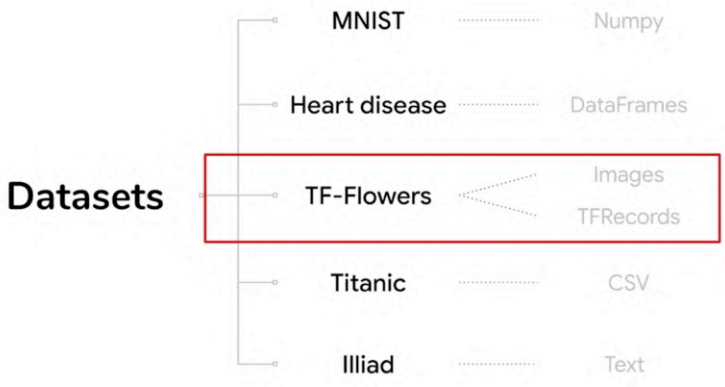
```
def make_generator():
  train_datagen = tf.keras.preprocessing.image.ImageDataGenerator(rescale=1. / 255,
                    rotation_range=20, zoom_range=[0.8, 1.2])

  train_generator = train_datagen.flow_from_directory(catsdogs,
                    target_size=(224, 224), class_mode='categorical',batch_size=32)

  return train_generator


train_generator = tf.data.Dataset.from_generator(
                          make_generator,(tf.float32, tf.uint8))
```

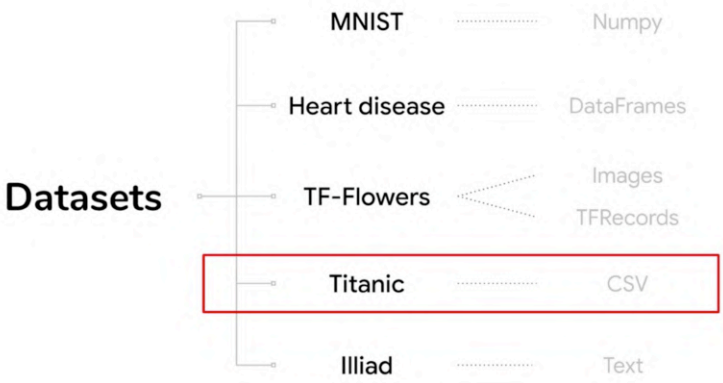
Generators — Keras ImageDataGenerator

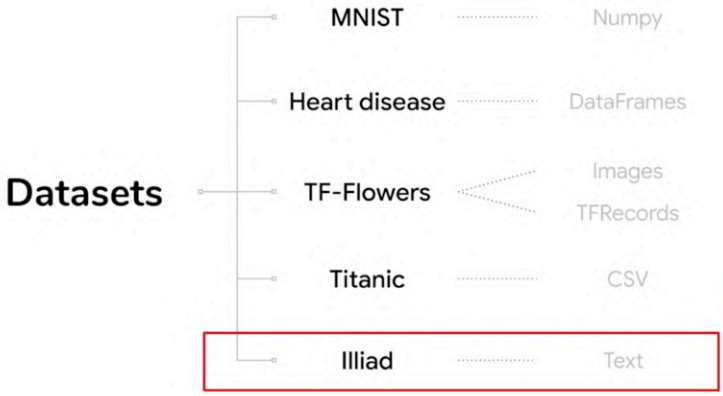
```
def make_generator():
  train_datagen = tf.keras.preprocessing.image.ImageDataGenerator(rescale=1. / 255,
                    rotation_range=20, zoom_range=[0.8, 1.2])

  train_generator = train_datagen.flow_from_directory(catsdogs,
                    target_size=(224, 224), class_mode='categorical',batch_size=32)

  return train_generator


train_generator = tf.data.Dataset.from_generator(
                          make_generator,(tf.float32, tf.uint8))
```


Datasets
MNIST
Numpy
Heart disease
DataFrames
TF-Flowers
Images
TFRecords
Titanic
CSV
Illiad
Text


Datasets
MNIST
Numpy
Heart disease
DataFrames
TF-Flowers
Images
TFRecords
Titanic
CSV
Illiad
Text


Datasets
MNIST
Numpy
Heart disease
DataFrames
TF-Flowers
Images
TFRecords
Titanic
CSV
Illiad
Text





Numpy | MNIST

```
# Fetch the numpy dataset
DATA_URL =
'https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz'
path = tf.keras.utils.get_file('mnist.npz', DATA_URL)

# Extract train, test sets
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']
```

So let's start with a dataset in dumpy formats, often with an NOPE file extension. In this case, it's super easy to load.

We can use keras.utils.get_file to specify the path, and then use np.load to load the data. It's super easy to then define the training and test samples and labels.

Numpy

MNIST

```
# Load them with tf.data
train_dataset = tf.data.Dataset.from_tensor_slices((train_examples, train_labels))
test_dataset = tf.data.Dataset.from_tensor_slices((test_examples, test_labels))

# Apply transformations like batch, shuffle to the dataset
train_dataset = train_dataset.shuffle(100).batch(64)
test_dataset = test_dataset.batch(64)
```

Numpy

Training on MNIST

When training, you can get the shape of the data using a numpy API

```
X, y = next(iter(train_dataset))
input_shape = X.numpy().shape[1:]

# Create a simple sequential model comprising of a Dense layer
model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=input_shape),
    ...
    tf.keras.layers.Dense(10, activation='softmax')])

model.compile(optimizer=tf.keras.optimizers.RMSprop(), loss=...,
metrics=...])

# Train the model
model.fit(train_dataset, epochs=10)
```

Numpy

Training on MNIST

When model.fitting, you can simply pass the training dataset.

```
X, y = next(iter(train_dataset))
input_shape = X.numpy().shape[1:]

# Create a simple sequential model comprising of a Dense layer
model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=input_shape),
    ...
    tf.keras.layers.Dense(10, activation='softmax')])

model.compile(optimizer=tf.keras.optimizers.RMSprop(), loss=...,
metrics=...])

# Train the model
model.fit(train_dataset, epochs=10)
```

Pandas

Identifying Heart Disease

```
csv_file = tf.keras.utils.get_file('heart.csv',
'https://storage.googleapis.com/applied-dl/heart.csv')

df = pd.read_csv(csv_file)
df['thal'] = pd.Categorical(df['thal'])
df['thal'] = df.thal.cat.codes

target = df.pop('target')
```

The target field at the end of the data table will be used as our label, the rest of the columns describe various parameters of the patient's heart condition.

To make the the thal column useable, we make use of Pandas categorical transformation to convert the string categories to discrete values.

Also, we pop the target from the dataframe so that we'll only be left with a dataframe of the feature set. Next, we'll see how these features and targets can be trained using a simple Keras model.

Pandas

Identifying Heart Disease

```
csv_file = tf.keras.utils.get_file('heart.csv', 'https://storage.googleapis.com/applied-dl/heart.csv')

df = pd.read_csv(csv_file)
df['thal'] = pd.Categorical(df['thal'])
df['thal'] = df.thal.cat.codes

target = df.pop('target')
```

	age	sex	cp	trestbps	chol	fbs	restecg	thalach	exang	oldpeak	slope	ca	thal
0	63	1	1	145	233	1	2	150	0	2.3	3	0	fixed
1	67	1	4	160	286	0	2	108	1	1.5	2	3	normal
2	67	1	4	120	229	0	2	129	1	2.6	2	2	reversible

Pandas

Training the model (Sequential)

```
dataset = tf.data.Dataset.from_tensor_slices((df.values, target.values))
train_dataset = dataset.shuffle(len(df)).batch(32)

model = tf.keras.Sequential([
    tf.keras.layers.Dense(10, activation='relu'),
    tf.keras.layers.Dense(10, activation='relu'),
    tf.keras.layers.Dense(1, activation='sigmoid')
])
model.compile(optimizer='adam',
              loss='binary_crossentropy',
              metrics=['accuracy'])

model.fit(train_dataset, epochs=15)
```

Pandas — Constructing the dataset for the Functional API

```
dict_slices = tf.data.Dataset.from_tensor_slices((df.to_dict('list'),
                                                  target.values)).batch(16)
```

We can do something more interesting using the Keras functional APIs.

The easiest way to preserve the column and structure of a Pandas DataFrame when used with tf.data is to convert it to a dictionary and then slice it

First, we'll create a dictionary of lists of all the features in the dataframe by calling to_dict list.

```
>>> for features, target in tfds.as_numpy(dict_slices.take(1)):
      for (feature, value), label in zip(features.items(), target):
         print('{} = {}\t Label = {}'.format(feature, value, label))

age  =  [63 67 67 37 41 56 62 57 63 53 57 56 56 44 52 57]     Label =  0
sex  =  [1 1 1 1 0 1 0 0 1 1 1 0 1 1 1 1]     Label =  1
cp  =  [1 4 4 3 2 2 4 4 4 4 4 2 3 2 3 3]     Label =  0
trestbps  =  [145 160 120 130 130 120 140 120 130 140 140 140 130 120 172 150]     Label =  0
chol  =  [233 286 229 250 204 236 268 354 254 203 192 294 256 263 199 168]     Label =  0
...
```

Pandas — Constructing the dataset for the Functional API

```
dict_slices = tf.data.Dataset.from_tensor_slices((df.to_dict('list'),
                                                  target.values)).batch(16)
```

let's inspect how the dataset looks like before loading the model with it. You'll see that each row in the dataset as a tuple of a feature dictionary and a label.

```
>>> for features, target in tfds.as_numpy(dict_slices.take(1)):
      for (feature, value), label in zip(features.items(), target):
         print('{} = {}\t Label = {}'.format(feature, value, label))

age  =  [63 67 67 37 41 56 62 57 63 53 57 56 56 44 52 57]     Label =  0
sex  =  [1 1 1 1 0 1 0 0 1 1 1 0 1 1 1 1]     Label =  1
cp  =  [1 4 4 3 2 2 4 4 4 4 4 2 3 2 3 3]     Label =  0
trestbps  =  [145 160 120 130 130 120 140 120 130 140 140 140 130 120 172 150]     Label =  0
chol  =  [233 286 229 250 204 236 268 354 254 203 192 294 256 263 199 168]     Label =  0
...
```

Pandas — Training the model (Functional)

Now we can easily create a series of Keras layers to define the model. But more importantly, we can see the shape of the inputs using an iterator across all of the keys.

```
# Constructing the inputs for all the dense features
inputs = {key: tf.keras.layers.Input(shape=(), name=key) for key in df.keys()}
x = tf.stack(list(inputs.values()), axis=-1)
x = tf.keras.layers.Dense(10, activation='relu')(x)

# The single output denoting the target's probability
output = tf.keras.layers.Dense(1, activation='sigmoid')(x)

model = tf.keras.Model(inputs=inputs, outputs=output)
model.compile(optimizer='adam',
              loss='binary_crossentropy',
              metrics=['accuracy'])

model.fit(dict_slices, epochs=15)
```

Pandas

Training the model (Functional)

Then you can train with a dict slices as the training dataset.

```
# Constructing the inputs for all the dense features
inputs = {key: tf.keras.layers.Input(shape=(), name=key) for key in df.keys()}
x = tf.stack(list(inputs.values()), axis=-1)
x = tf.keras.layers.Dense(10, activation='relu')(x)

# The single output denoting the target's probability
output = tf.keras.layers.Dense(1, activation='sigmoid')(x)

model = tf.keras.Model(inputs=inputs, outputs=output)
model.compile(optimizer='adam',
              loss='binary_crossentropy',
              metrics=['accuracy'])

model.fit(dict_slices, epochs=15)
```

Images

Classifying species of flowers

```
DATA_URL = '[insert URL here]'
data_root_orig = tf.keras.utils.get_file(origin=DATA_URL,
                                         fname='flower_photos', untar=True)
data_root = pathlib.Path(data_root_orig)

# Load all the file paths in the directory
all_image_paths = list(data_root.glob('*/*'))
all_image_paths = [str(path) for path in all_image_paths]

# Gather the list of labels and create a labelmap
label_names = sorted(item.name for item in data_root.glob('*/') if item.is_dir())
label_to_index = dict((name, index) for index, name in enumerate(label_names))

# Use the label map to fetch all categorical labels
all_image_labels = [label_to_index[pathlib.Path(path).parent.name]
                    for path in all_image_paths]
```

Create a list of all of the images with their paths

Images

Classifying species of flowers

```
DATA_URL = '[insert URL here]'
data_root_orig = tf.keras.utils.get_file(origin=DATA_URL,
                                         fname='flower_photos', untar=True)
data_root = pathlib.Path(data_root_orig)

# Load all the file paths in the directory
all_image_paths = list(data_root.glob('*/*'))
all_image_paths = [str(path) for path in all_image_paths]

# Gather the list of labels and create a labelmap
label_names = sorted(item.name for item in data_root.glob('*/') if item.is_dir())
label_to_index = dict((name, index) for index, name in enumerate(label_names))

# Use the label map to fetch all categorical labels
all_image_labels = [label_to_index[pathlib.Path(path).parent.name]
                    for path in all_image_paths]
```

Sort through them extracting the names of the directories and creating an index of the directories to get a set of index labels.

Images — Classifying species of flowers

```
DATA_URL = '[insert URL here]'
data_root_orig = tf.keras.utils.get_file(origin=DATA_URL,
                                         fname='flower_photos', untar=True)
data_root = pathlib.Path(data_root_orig)

# Load all the file paths in the directory
all_image_paths = list(data_root.glob('*/*'))
all_image_paths = [str(path) for path in all_image_paths]

# Gather the list of labels and create a labelmap
label_names = sorted(item.name for item in data_root.glob('*/') if item.is_dir())
label_to_index = dict((name, index) for index, name in enumerate(label_names))

# Use the label map to fetch all categorical labels
all_image_labels = [label_to_index[pathlib.Path(path).parent.name]
                    for path in all_image_paths]
```

Use the labels to create a categorical label set.

Images — Preprocessing and creating the dataset

```
path_ds = tf.data.Dataset.from_tensor_slices(all_image_paths)
label_ds = tf.data.Dataset.from_tensor_slices(all_image_labels)

def preprocess_image(path):
  image = tf.io.read_file(path)
  image = tf.image.decode_jpeg(image, channels=3)
  image = tf.image.resize(image, [192, 192])
  image /= 255.0  # normalize to [0,1] range
  return image

image_ds = path_ds.map(preprocess_image)
image_label_ds = tf.data.Dataset.zip((image_ds, label_ds))
```

Now that we have the files, we can create a data set from them and do pre-processing on it if we want to.

So we can use from tensor slices on the image pads and labels list we created earlier to get path and label datasets.

Images — Preprocessing and creating the dataset

```
path_ds = tf.data.Dataset.from_tensor_slices(all_image_paths)
label_ds = tf.data.Dataset.from_tensor_slices(all_image_labels)

def preprocess_image(path):
  image = tf.io.read_file(path)
  image = tf.image.decode_jpeg(image, channels=3)
  image = tf.image.resize(image, [192, 192])
  image /= 255.0  # normalize to [0,1] range
  return image

image_ds = path_ds.map(preprocess_image)
image_label_ds = tf.data.Dataset.zip((image_ds, label_ds))
```

We can then create an image data set by mapping the results of pre-processing the image. Where the pre-process is the helper function above that does some form of pre-processing. In this case, it decodes the jpeg, resizes it to 192x192 and then normalizes it

Images

Preprocessing and creating the dataset

```
path_ds = tf.data.Dataset.from_tensor_slices(all_image_paths)
label_ds = tf.data.Dataset.from_tensor_slices(all_image_labels)

def preprocess_image(path):
  image = tf.io.read_file(path)
  image = tf.image.decode_jpeg(image, channels=3)
  image = tf.image.resize(image, [192, 192])
  image /= 255.0  # normalize to [0,1] range
  return image

image_ds = path_ds.map(preprocess_image)
image_label_ds = tf.data.Dataset.zip((image_ds, label_ds))
```

We can also get a dataset of our image labels

Images

Training the model

If we're doing some training, we can do fancy things like shuffling the dataset, batching it and figuring out the appropriate steps per Epoch based on the batch size. Then when fitting, it's as easy as passing the data set to model.fit.

```
BATCH_SIZE = 32
ds = image_label_ds.shuffle(
            buffer_size=len(all_image_paths)).repeat().batch(BATCH_SIZE)

steps_per_epoch=tf.math.ceil(len(all_image_paths) / BATCH_SIZE).numpy()

model.fit(ds, epochs=1, steps_per_epoch=steps_per_epoch)
```

CSV

Predicting survivors with Titanic

```
train_file_path = tf.keras.utils.get_file(
    "train.csv","https://storage.googleapis.com/tf-datasets/titanic/train.csv")
test_file_path = tf.keras.utils.get_file(
    "Eval.csv", "https://storage.googleapis.com/tf-datasets/titanic/eval.csv")

def get_dataset(file_path, **kwargs):
  dataset = tf.data.experimental.make_csv_dataset(
      file_path,
      batch_size=5, # Artificially small to make examples easier to show.
      label_name='survived',
      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)
```

Make CSV data set will take a CSV file and turn it into a data set for us

CSV

Predicting survivors with Titanic

```
train_file_path = tf.keras.utils.get_file(
    "train.csv","https://storage.googleapis.com/tf-datasets/titanic/train.csv")
test_file_path = tf.keras.utils.get_file(
    "Eval.csv", "https://storage.googleapis.com/tf-datasets/titanic/eval.csv")

def get_dataset(file_path, **kwargs):
  dataset = tf.data.experimental.make_csv_dataset(
      file_path,
      batch_size=5, # Artificially small to make examples easier to show.
      label_name='survived',
      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)
```

Parameters within this can help shape the dataset. So for example here, we have a batch size of 5 on the new dataset.

CSV

What happens after loading the CSV?

We'll use a helper function called show batch. Which as you can see demonstrated here, shows the feature key and number of samples for that key.

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

```
>>> show_batch(get_dataset(train_file_path))
sex                 : [b'female' b'female' b'female' b'male' b'male']
age                 : [40. 28. 52. 50. 34.]
n_siblings_spouses  : [0 0 1 0 1]
parch               : [0 0 0 0 0]
fare                : [13.      7.75   78.2667 13.     21.    ]
class               : [b'Second' b'Third' b'First' b'Second' b'Second']
deck                : [b'unknown' b'unknown' b'D' b'unknown' b'unknown']
embark_town         : [b'Southampton' b'Queenstown' b'Cherbourg' b'Southampton' ..]
alone               : [b'y' b'y' b'n' b'y' b'n']
```

CSV

Getting data from named columns

Columns in a CSV can be named. So if we create a list of columns that were interested in, we can create the dataset using only them. So when we show our batch, we can see that our dataset uses only those columns from the CSV and ignores the rest. If the CSV doesn't have column names,

```
CSV_COLUMNS = ['survived', 'sex', 'age', 'n_siblings_spouses', 'parch', 'fare',
'class', 'deck', 'embark_town', 'alone']

temp_dataset = get_dataset(train_file_path, column_names=CSV_COLUMNS)

>>> show_batch(temp_dataset)

sex                 : [b'female' b'male' b'male' b'male' b'male']
age                 : [15. 29. 49. 35. 22.]
n_siblings_spouses  : [1 1 1 0 0]
parch               : [0 0 1 0 0]
fare                : [ 14.4542  21.     110.8833   7.125    7.125 ]
class               : [b'Third' b'Second' b'First' b'Third' b'Third']
deck                : [b'unknown' b'unknown' b'C' b'unknown' b'unknown']
embark_town         : [b'Cherbourg' b'Southampton' b'Cherbourg' b'Southampton'..]
alone               : [b'n' b'n' b'n' b'y' b'y']
```

CSV — Omitting some columns

Of course, if you don't want all the columns, you can use smaller lists of columns to select like this. Simply create the list of columns that you want and pass them into the get data set method in the select columns parameter.

```
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                 : [60. 34. 28. 40. 28.]
n_siblings_spouses  : [1 1 1 0 0]
class               : [b'Second' b'Third' b'Third' b'First' b'Third']
deck                : [b'unknown' b'unknown' b'unknown' b'B' b'unknown']
alone               : [b'n' b'n' b'n' b'y' b'y']
```

CSV — Extracting features

```
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)

# Function that will pack together all the columns:
def pack(features, label):
  return tf.stack(list(features.values()), axis=-1), label

packed_dataset = temp_dataset.map(pack)
```

One issue to consider is that a dataset can contain multiple types. So for example, there may be strings, Boolean's, numeric values and more. They're typically stored as strings and cast into the correct data type after reading. But before you feed the data into the model, you will want them in a fixed length vector. For that, you can use feature columns as before. So in order to pack the features, we'll get are required set of columns as before but this time we'll set default values.

CSV — Extracting features

```
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)

# Function that will pack together all the columns:
def pack(features, label):
  return tf.stack(list(features.values()), axis=-1), label

packed_dataset = temp_dataset.map(pack)
```

Once we've done that, a helper function called pack will use tf.stack to give us a pact dataset. Where there's no sparseness because we use defaults and it's regularly shaped because of tf.stack.

CSV — Packing numeric features

```
NUMERIC_FEATURES = ['age','n_siblings_spouses','parch', 'fare']

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

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

Similarly, we can stack numeric features. So for example, here is a list of numeric features in this dataset and a class that will allow us to pack them.

Within the CSV, the value is stored as a string so we cast it to float upon reading it.

CSV — Showing packed features

```
>>> show_batch(packed_train_data)
sex          : [b'male' b'male'  ...]
class        : [b'First' b'Third'  ...]
deck         : [b'unknown' b'unknown'  ...]
embark_town  : [b'Cherbourg' b'Southampton'  ...]
alone        : [b'n' b'y'     ...]
numeric      : [[28.      1.   ...]
                [49.      0.   ...]
                [27.      0.   ...]
                [0.83     0.   ...]
                [28.      0.   ...]]
```

Again, we can use tf.stack to pack it and flatten it out. If we show our batch, we can now see the pack features where we've parsed out the CSV data, converted it to the appropriate data type and then flattened it out.

CSV — Normalizing features

```
NUMERIC_FEATURES = ['age','n_siblings_spouses','parch', 'fare']

def normalize_numeric_data(data, mean, std):
  # Center the data
  return (data-mean)/std

desc = pd.read_csv(train_file_path)[NUMERIC_FEATURES].describe()

MEAN, STD = np.array(desc.T['mean']), np.array(desc.T['std'])

normalizer = functools.partial(normalize_numeric_data,
                               mean=MEAN,
                               std=STD)

numeric_column = tf.feature_column.numeric_column(
                                  'numeric',
                                  normalizer_fn=normalizer,
                                  shape=[len(NUMERIC_FEATURES)])
```

Next up is the task of normalizing. Given that our numeric data may be in vastly different ranges, a good normalization function is needed. We can't just divide by 255 like is commonly done when normalizing for grayscale images like mnist and fashion mnist.

One of the simplest ways to do this is to use pandas to read the data and numpy to calculate the mean and standard deviation of the data.

CSV

Normalizing features

```
NUMERIC_FEATURES = ['age','n_siblings_spouses','parch', 'fare']

def normalize_numeric_data(data, mean, std):
  # Center the data
  return (data-mean)/std

desc = pd.read_csv(train_file_path)[NUMERIC_FEATURES].describe()

MEAN, STD = np.array(desc.T['mean']), np.array(desc.T['std'])

normalizer = functools.partial(normalize_numeric_data,
                               mean=MEAN,
                               std=STD)

numeric_column = tf.feature_column.numeric_column(
                                  'numeric',
                                  normalizer_fn=normalizer,
                                  shape=[len(NUMERIC_FEATURES)])
```

Normalization is then quite easy through subtracting the mean and dividing by the standard deviation and we can encapsulate this in a function.

CSV

Normalizing features

```
NUMERIC_FEATURES = ['age','n_siblings_spouses','parch', 'fare']

def normalize_numeric_data(data, mean, std):
  # Center the data
  return (data-mean)/std

desc = pd.read_csv(train_file_path)[NUMERIC_FEATURES].describe()

MEAN, STD = np.array(desc.T['mean']), np.array(desc.T['std'])

normalizer = functools.partial(normalize_numeric_data,
                               mean=MEAN,
                               std=STD)

numeric_column = tf.feature_column.numeric_column(
                                  'numeric',
                                  normalizer_fn=normalizer,
                                  shape=[len(NUMERIC_FEATURES)])
```

Declare the normalizer to use this and passing that to tf.feature column, numeric column to create a numeric column that is normalized using that normalizer.

CSV

Now for the categorical features

```
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']
}
```

So that's string and numeric features, but what about categorical ones? For example in the Titanic dataset, you have the class of service, the deck, the town that the person embarked on the ship and much more. The indicator column type is your friend here.

So first, you'll create a categorical feature column with a vocabulary list telling it the key and the list. So in this case, we'll use the classes the key and first second and third the options for class in the list. Then, all we have to do is specify that we want an indicator column built off of that.

```
cat_feature_col = tf.feature_column.categorical_column_with_vocabulary_list(
                    key='class',
                    vocabulary_list=['First', 'Second', 'Third'])

categorical_column = tf.feature_column.indicator_column(cat_feature_col))
```

CSV

Training the model

```
dense_features= tf.keras.layers.DenseFeatures(categorical_columns+numeric_columns)

model = tf.keras.Sequential([
  dense_features,
  tf.keras.layers.Dense(128, activation='relu'),
  tf.keras.layers.Dense(128, activation='relu'),
  tf.keras.layers.Dense(1, activation='sigmoid'),
])

model.compile(
    loss='binary_crossentropy',
    optimizer='adam',
    metrics=['accuracy'])

model.fit(packed_train_data, epochs=20)
```

Let's now look at how you feed these into a model for training. Once you've gone through this process, you'll have a bunch of categorical columns and numeric columns. You can now cast these into a set of dense features using keras dense features and you simply add the two sets and pass them in as a parameter.

CSV

Training the model

```
dense_features= tf.keras.layers.DenseFeatures(categorical_columns+numeric_columns)

model = tf.keras.Sequential([
  dense_features,
  tf.keras.layers.Dense(128, activation='relu'),
  tf.keras.layers.Dense(128, activation='relu'),
  tf.keras.layers.Dense(1, activation='sigmoid'),
])

model.compile(
    loss='binary_crossentropy',
    optimizer='adam',
    metrics=['accuracy'])

model.fit(packed_train_data, epochs=20)
```

These dense features can be the first layer in your sequential after which creating the code for the model should look very familiar.

Text

Identifying translators of a work

```
DIRECTORY_URL = 'https://storage.googleapis.com/download.tensorflow.org/data/illiad/'
FILE_NAMES = ['cowper.txt', 'derby.txt', 'butler.txt']

def labeler(example, index):
  return example, tf.cast(index, tf.int64)

labeled_data_sets = []
for i, file_name in enumerate(FILE_NAMES):
  file_path = tf.keras.utils.get_file(name, origin=DIRECTORY_URL+file_name)
  lines_dataset = tf.data.TextLineDataset(file_path)
  labeled_dataset = lines_dataset.map(lambda ex: labeler(ex, i))
  labeled_data_sets.append(labeled_dataset)
```

What would it take to train a classifier to recognize which translation a given line of text comes from?

Well, to do this, we have to start by reading all of the text. We'll do this by using tf.data.TextLineDataset to read the text in. Because this is done as a loop in an enumerator, then we'll label each dataset automatically, i.e, labels will be zero for Cowper, one for Darby, and two for Butler.

Text

Preparing the dataset

```
dataset = labeled_data_sets[0]
for labeled_dataset in labeled_data_sets[1:]:
  dataset = dataset.concatenate(labeled_dataset)

dataset = dataset.shuffle(buffer_size=50000)

>>> for ex in dataset.take(5):
      print(ex[0].numpy(), ex[1].numpy())
b"Eight barbed arrows have I shot e'en now," 1
b'In thy own band; the Achaians shall for him,' 0
b"Upon their well-mann'd ships, should Heaven vouchsafe" 1
b'He shall not cozen me! Of him, enough!' 1
b'Turns flying, marks him with a steadfast eye,' 0
```

Once we've done that, it's pretty easy for us to then concatenate all of these into one large dataset. If we shuffle that dataset and print some outputs from it, we'll see that we have lines of labeled text.

Text

Text encoding

```
tokenizer = tfds.features.text.Tokenizer()

vocabulary_set = set()
for text_tensor, _ in all_labeled_data:
  some_tokens = tokenizer.tokenize(text_tensor.numpy())
  vocabulary_set.update(some_tokens)

vocab_size = len(vocabulary_set)
>>> vocab_size
17178
```

However, we train models using numeric values instead of textual ones. So the words needs to be encoded into numbers. The easiest way to do this is using a tokenizer, which is built into tfds.

This simply goes through the dataset, replacing words with numbers and creating a dictionary of mappings. So for example, the word the can become the token one. Then every instance of the will become one in the tokenized dataset.

Text

Encode an example

```
# Show one of the labeled data
original_text = next(iter(all_labeled_data))[0].numpy()

# Create an text encoder with a fixed vocabulary set
encoder = tfds.features.text.TokenTextEncoder(vocabulary_set)

# Encode an example
encoded_text = encoder.encode(original_text)

Original text  b"As honour's meed, the mighty monarch gave."
Encoded text  [16814, 4289, 11591, 15925, 177, 10357, 11207, 16715]
```

So to encode an example with a dictionary that was created, we just use a token text encoder with it to create an encoder. We can then encode a single sentence and see that it's words are replaced by numeric tokens.

Text

Encoding all the examples

```
def encode(text_tensor, label):
  encoded_text = encoder.encode(text_tensor.numpy())
  return encoded_text, label

def encode_map_fn(text, label):
  return tf.py_function(encode, inp=[text, label],
                        Tout=(tf.int64, tf.int64))

all_encoded_data = all_labeled_data.map(encode_map_fn)
```

If we want to encode all of the sentences and keep track of their labels, we can write a helper function called encode which does this for us, and then use that in a mapping function. A mapping function in a dataset applies a function across each element in the dataset and returns a new dataset containing the transformed elements in the same order as they appeared in the original dataset. Thus, we can use the mapping function to give us a simple way to pass our custom encode function to the dataset and create a new one containing the tokenized text.

Text

Prepare the dataset

```
BUFFER_SIZE = 50000
BATCH_SIZE = 64
TAKE_SIZE = 5000

train_data = all_encoded_data.skip(TAKE_SIZE).shuffle(BUFFER_SIZE)
train_data = train_data.padded_batch(BATCH_SIZE, padded_shapes=([-1],[]))

test_data = all_encoded_data.take(TAKE_SIZE)
test_data = test_data.padded_batch(BATCH_SIZE, padded_shapes=([-1],[]))
```

Now comes the task of preparing the dataset. For example, a shuffling it and splitting it into training and text datasets. One thing with text datasets is that because they're made up of sentences which aren't always the same length, will pad using padded batch so that they all end up the same size, which is typically the size of the longest sentence.

Text

Training the model

```
model = tf.keras.Sequential([
    tf.keras.layers.Embedding(vocab_size, 64),
    tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(64)),
    tf.keras.Sequential([
        tf.keras.layers.Dense(units, activation='relu') for units in [64, 64]
    ]),
    tf.keras.layers.Dense(3, activation='softmax')
])

model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

model.fit(train_data, epochs=3)
```

Then training the model is done just like before. We pass the training data to the model fit.