Getting Data into TensorFlow Estimator Models

Robert Thas John Aug 26, 2018 · 6 min read

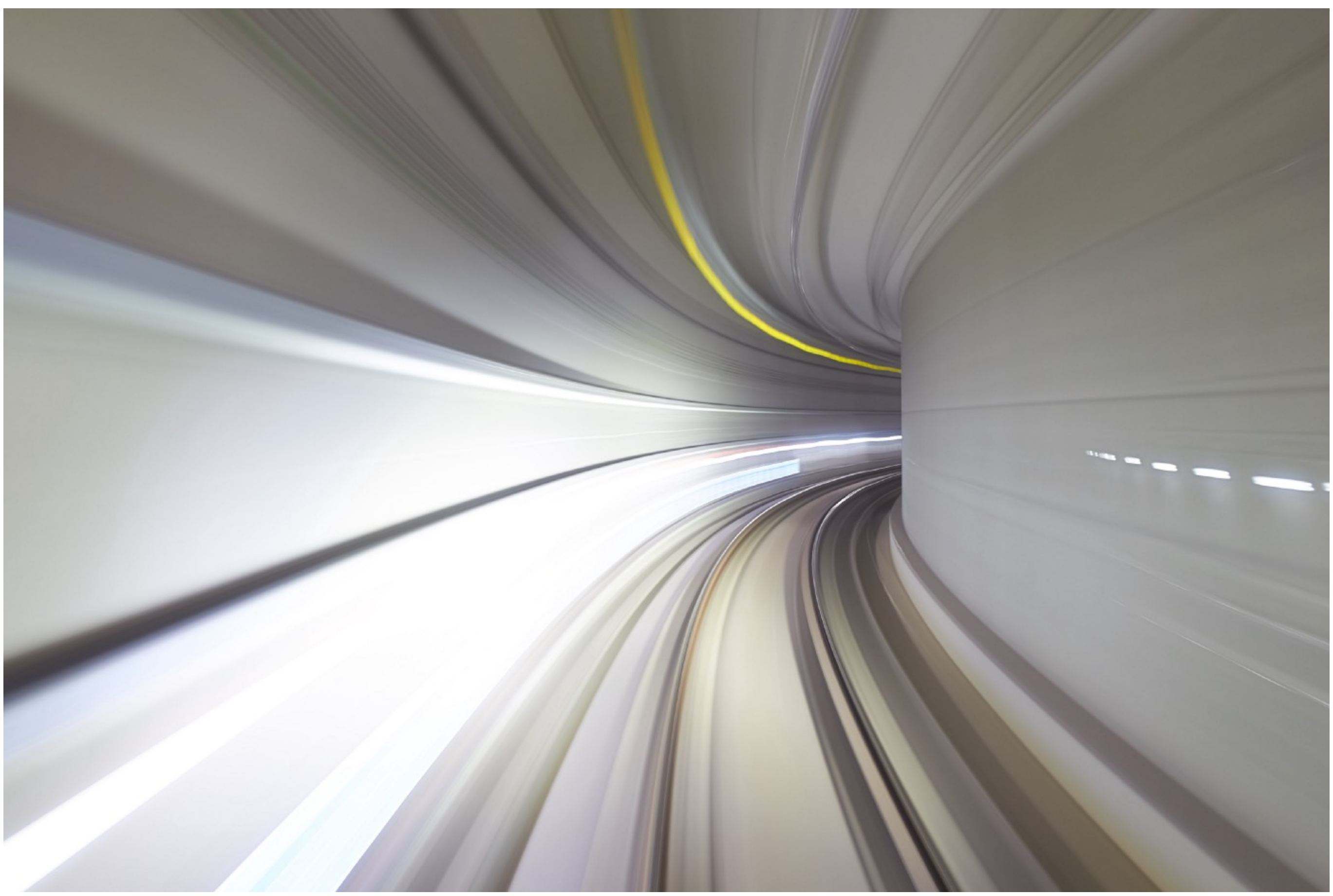

Photo by Mathew Schwartz on Unsplash

Machine Learning is all about the quantity and quality of your data. The said data is usually made available in a variety of sources:

- Text files (CSV, TSV, Excel)
- Databases
- Streaming Sources

Text files are made available by some person or persons who extract the data from another source, but wish to save you the stress of extracting the data yourself. The data could be in one or more files, with or without headers.

TensorFlow estimators work with input functions. The signature of an input function returns a tuple of features and labels. Features are a dictionary of feature names and numeric value arrays. Labels are an array of values. Some management needs to happen, such as shuffling the data, and returning it in batches. The approach you take determines how much effort you need to put in.

Let's start with the simple option. If you have your data in one file, which you are able to read completely into memory (so-called toy examples), and the file is in text-delimited format (CSV, TSV, etc), the amount of effort required is minimal. You can read your files in with numpy or pandas, as is commonly the case.

As a reminder, when you work with `tf.estimator` API, you need to pass in an input function during training. This is the function signature for training:

```
train(
    input_fn,
    hooks=None,
    steps=None,
    max_steps=None,
    saving_listeners=None
)
```

Our focus is on `input_fn` ! We will work with the popular Boston Housing data which is hosted here.

If you have your data in numpy format, you can use `tf.estimator.inputs.numpy_input_function` to get your data in. First you need to define a dictionary for your features:

```
# extract numpy data from a DataFrame
crim = train_df['crim'].values
zn = train_df['zn'].values
indus = train_df['indus'].values
chas = train_df['chas'].values
nox = train_df['nox'].values
rm = train_df['rm'].values
age = train_df['age'].values
dis = train_df['dis'].values
rad = train_df['rad'].values
tax = train_df['tax'].values
ptratio = train_df['ptratio'].values
black = train_df['black'].values
lstat = train_df['lstat'].values
medv = train_df['medv'].values

# create a dictionary
x_dict = {
    'crim': crim,
    'zn': zn,
    'indus': indus,
    'chas': chas,
    'nox': nox,
    'rm': rm,
    'age': age,
    'dis': dis,
    'rad': rad,
    'tax': tax,
    'ptratio': ptratio,
    'black': black,
    'lstat': lstat
}
```

With our dictionary in place, we may proceed to define our input function.

```
def np_training_input_fn(x, y):
  return tf.estimator.inputs.numpy_input_fn(
      x= x,
      y= y,
      batch_size= 32,
      num_epochs= 5, # this way you can leave out steps from
training
```

```
        shuffle= True,
        queue_capacity= 5000
    )
```

In our function, we pass in x, which is our dictionary, and y, which is our label. We can also pass in our batch size, number of epochs, and whether or not to shuffle the data. Please note that you always want to shuffle your data. The batch size is a hyper parameter that you should file empirically. The number of epochs is how many times you would like to go over your data. For training, set any number. For test, set this to 1.

Before creating your estimator, you will need feature columns.

```
feature_cols = [tf.feature_column.numeric_column(k) for k in
x_dict.keys()]

lin_model =
tf.estimator.LinearRegressor(feature_columns=feature_cols)

lin_model.train(np_training_input_fn(x_dict, medv), steps=10)
```

You can leave out steps, so the training uses the epochs specified in your training input function, or specify the number of steps to use for training. That's all for numpy input.

For a DataFrame, you would proceed to define the input function as follows:

```
def pd_input_fn(df, y_label):
  return tf.estimator.inputs.pandas_input_fn(
      x=df,
      y=df[y_label],
      batch_size = 32,
      num_epochs = 5,
      shuffle = True,
      queue_capacity = 1000,
      num_threads = 1
  )
```

Note that in the above method, we proceed to pass in our DataFrame, complete with the label in it. If the label is not in what you pass to `x`, you will get an error. You pass a series to `y`. The other parameters are the same as when you deal with numpy.

The model is treated the same going forward. You create the model and specify the feature columns. You then proceed to train the mode.

```
lin_model = 
tf.estimator.LinearRegressor(feature_columns=feature_cols)

lin_model.train(pd_input_fn(train_df, 'medv'), steps=10)
```

It's all well and good when you can read your data into memory. But, what happens when you can't. What happens when your training dataset is 100GB?

The good news is such a dataset will normally be produced by a distributed system, so your files will be `sharded`. That means the data will be stored in different files with names like `data-0001-of-1000`.

If you have never dealt with Big Data, your first thought might be to use `glob`. Do not do that unless you know that you are dealing with a toy example. You will exhaust your memory and training will stop.

These types of files normally do not have headers, and that is a good thing. You will start by defining a list of column names which should be in the order in which your columns exist in the files. Secondly, define a label column. Finally, define a list of defaults so you can handle missing values when you encounter them during reading.

```
CSV_COLUMNS = ['medv', 'crim', 'zn', 'lstat', 'tax', 'rad', 'chas', 
'nox', 'indus', 'ptratio', 'age', 'black', 'rm', 'dis']
LABEL_COLUMN = 'medv'
DEFAULTS = [[0.0], [0.0], [0.0], [0.0], [0.0], [0.0], [0.0], [0.0], 
[0.0], [0.0], [0.0], [0.0], [0.0], [0.0]]
```

Next, we define a function to read in text data and return our format in the same way that our earlier functions were handling them. One advantage of the way the function is created is that it can handle wildcards, such as `data-*` .

```
def read_dataset(filename, mode, batch_size = 512):
  def _input_fn():
    def decode_csv(value_column):
      columns = tf.decode_csv(value_column, record_defaults =
DEFAULTS)
      features = dict(zip(CSV_COLUMNS, columns))
      label = features.pop(LABEL_COLUMN)
      return features, label

    # Create list of files that match pattern
    file_list = tf.gfile.Glob(filename)

    # Create dataset from file list
    dataset = tf.data.TextLineDataset(file_list).map(decode_csv)
    if mode == tf.estimator.ModeKeys.TRAIN:
        num_epochs = None # indefinitely
        dataset = dataset.shuffle(buffer_size = 10 * batch_size)
    else:
        num_epochs = 1 # end-of-input after this

    dataset = dataset.repeat(num_epochs).batch(batch_size)
    return dataset.make_one_shot_iterator().get_next()
  return _input_fn
```

The function takes in three parameters: a pattern so we can match multiple files, a mode (training or evaluation), and a batch size. Notice that `read_dataset` returns a function. We have called that function `_input_fn` . Inside this function, we have a function called `decode_csv` that will create a dictionary, extract a series, and return both in the tuple format we mentioned at the beginning of this article.

Secondly, our function creates a list of file names using `glob` . Yes, `glob` is still used, but we don't pass the result to a `pandas.read_csv()` . Instead, meet `tf.data.TextLineDataset()` . It takes three parameters: a list of file names, the compression format (none, ZLIB, or GZIP), and a buffer size. The primary difference between `read_csv` and `TextLineDataset` is that the former reads the contents into memory (we can read in batches), while the latter returns an `Iterator` .

So, our function creates a dataset using `TextLineDataset` by calling the `map` function, passing in `decode_csv`. The next thing it does is check whether or not we are in training mode. If we are not, our number of epochs is set to 1. If we are, it is set to however many epochs we would like. Our training dataset is also shuffled. Our dataset is then set to repeat the number of epochs we would like, and configured for our batch size.

Finally, we return a one-shot iterator, and call `get_next()`. All of this work is handled behind the scenes by the functions we saw earlier. We can create our training, evaluation, and test input functions using the following approach:

```
def get_train():
  return read_dataset('./train-.*', mode =
tf.estimator.ModeKeys.TRAIN)

def get_valid():
  return read_dataset('./valid.csv', mode =
tf.estimator.ModeKeys.EVAL)

def get_test():
  return read_dataset('./test.csv', mode =
tf.estimator.ModeKeys.EVAL)
```

The rest of the process is exactly the same as we have seen. We can create our estimator and train it as usual.

For real projects, you will start by reading in one of your training files using pandas and `tf.estimator.inputs`. However, to use all of your files in training, you will want to use `tf.data.TextLineDataset`.

Happy coding.

Get this newsletter

By signing up, you will create a Medium account if you don't already have one. Review our Privacy Policy for more information about our privacy practices.

Data Science TensorFlow Big Data