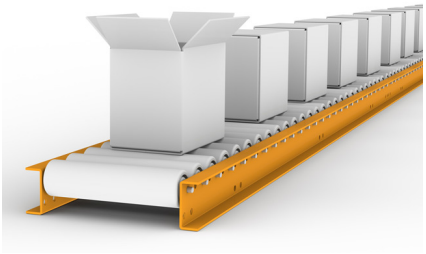
THE DATA SCIENCE LAB

How to Create and Use a PyTorch DataLoader

Dr. James McCaffrey of Microsoft Research provides a full code sample and screenshots to explain how to create and use PyTorch Dataset and DataLoader objects, used to serve up training or test data in order to train a PyTorch neural network.

By James McCaffrey 09/10/2020

GET CODE DOWNLOAD



In order to train a PyTorch neural network you must write code to read training data into memory, convert the data to PyTorch tensors, and serve the data up in batches. This task is not trivial and is often one of the biggest roadblocks for people who are new to PyTorch.

In the early days of PyTorch, you had to write completely custom code for data loading. Now however, the vast majority of PyTorch systems I've seen (and created myself) use the PyTorch Dataset and DataLoader interfaces to serve up training or test data. Briefly, a Dataset object loads training or test data into memory, and a DataLoader object fetches data from a Dataset and serves the data up in batches.

You must write code to create a Dataset that matches your data and problem scenario; no two Dataset implementations are exactly the same. On the other hand, a DataLoader object is used mostly the same no matter which Dataset object it's associated with. For example:

```
class MyDataSet(T.utils.data.Dataset):
  # implement custom code to load data here

my_ds = MyDataset("my_train_data.txt")
```

```
my_ldr = torch.utils.data.DataLoader(my_ds, 10, True)
for (idx, batch) in enumerate(my_ldr):
  . . .
```

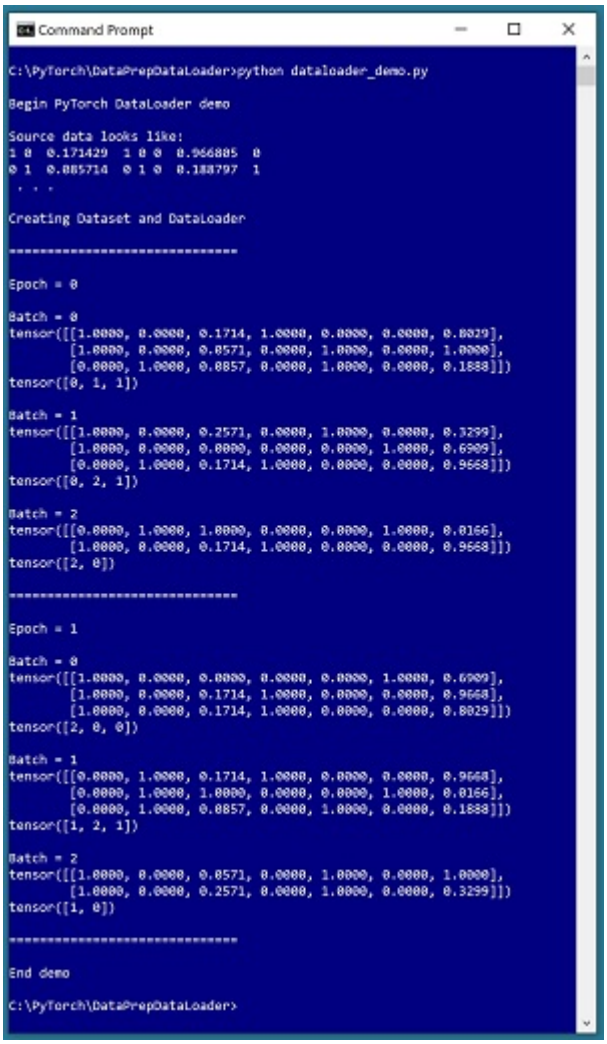
The code fragment shows you must implement a Dataset class yourself. Then you create a Dataset instance and pass it to a DataLoader constructor. The DataLoader object serves up batches of data, in this case with batch size = 10 training items in a random (True) order.

This article explains how to create and use PyTorch Dataset and DataLoader objects. A good way to see where this article is headed is to take a look at the screenshot of a demo program in **Figure 1**. The source data is a tiny 8-item file. Each line represents a person: sex (male = 1 0, female = 0 1), normalized age, region (east = 1 0 0, west = 0 1 0, central = 0 0 1), normalized income, and political leaning (conservative = 0, moderate = 1, liberal = 2). The goal of the demo is to serve up data in batches where the dependent variable to predict is political leaning, and the other variables are the predictors.

The 8-item source data is stored in a tab-delimited file named people_train.txt and looks like:

```
1 0  0.171429  1 0 0  0.966805  0
0 1  0.085714  0 1 0  0.188797  1
1 0  0.000000  0 0 1  0.690871  2
1 0  0.057143  0 1 0  1.000000  1
0 1  1.000000  0 0 1  0.016598  2
1 0  0.171429  1 0 0  0.802905  0
0 1  0.171429  1 0 0  0.966805  1
1 0  0.257143  0 1 0  0.329876  0
```


Behind the scenes, the demo loads data into memory using a custom Dataset object, and then serves the data up in randomly selected batches of size 3 rows/items. Because the source data has 8 lines, the first two batches have 3 data items, but the last batch has 2 items. The demo processes the source data twice, in other words, two epochs.



[Click on image for larger view.]

Figure 1: PyTorch DataLoader Demo

This article assumes you have intermediate or better skill with a C-family programming language. The demo program is coded using Python, which is used by PyTorch and which is essentially the primary language for deep neural networks. The complete source code for the demo program is presented in this article. The source code and source data are also available in the file download that accompanies this article.

The Demo Program

The demo program, with a few minor edits to save space, is presented in **Listing 1**. I indent my Python programs using two spaces, rather than the more common four spaces or a tab character, as a matter of personal preference.

Listing 1: DataLoader Demo Program

```
# dataloader_demo.py
# PyTorch 1.5.0-CPU  Anaconda3-2020.02
# Python 3.7.6  Windows 10

import numpy as np
import torch as T
device = T.device("cpu")  # to Tensor or Module

# -----------------------------------------------------

# predictors and label in same file
# data has been normalized and encoded like:
#   sex     age      region   income    politic
#   [0]     [2]       [3]      [6]       [7]
#   1 0   0.057143   0 1 0    0.690871    2
```

The execution of the demo program begins with:

```
def main():
  print("\nBegin PyTorch DataLoader demo ")

  # 0. miscellaneous prep
  T.manual_seed(0)
  np.random.seed(0)
  . . .
```

In almost all PyTorch programs, it's a good idea to set the system random number generator seed values so that your results will be reproducible. Unfortunately, because of execution across multiple processes, sometimes your results are not reproducible even if you set the random generator seeds. But if you don't set the seeds, your results will almost certainly not be reproducible.

Next, a Dataset and a DataLoader object are created:

```
  train_file = ".\\people_train.txt"
  train_ds = PeopleDataset(train_file, num_rows=8)
```

```
bat_size = 3
train_ldr = T.utils.data.DataLoader(train_ds,
  batch_size=bat_size, shuffle=True)
```

The custom PeopleDataset object constructor accepts a path to the training data, and a num_rows parameter in case you want to load just part of a very large data file during system development.

The built-in DataLoader class definition is housed in the torch.utils.data module. The class constructor has one required parameter, the Dataset that holds the data. There are 10 optional parameters. The demo specifies values for just the batch_size and shuffle parameters, and therefore uses the default values for the other 8 optional parameters.

The demo concludes by using the DataLoader to iterate through the source data:

```
for epoch in range(2):
  print("\n==============================\n")
  print("Epoch = " + str(epoch))
  for (batch_idx, batch) in enumerate(train_ldr):
    print("\nBatch = " + str(batch_idx))
    X = batch['predictors']  # [3,7]
    Y = batch['political']   # [3]
    print(X)
    print(Y)
```


In neural network terminology, an epoch is one pass through all source data. The DataLoader class is designed so that it can be iterated using the enumerate() function, which returns a tuple with the current batch zero-based index value, and the actual batch of data. There is a tight coupling between a Dataset and its associated DataLoader, meaning you have to know the names of the keys used for the predictor values and the dependent variable values. In this case the two keys are "predictors" and "political."

Implementing a Dataset Class

You have a lot of flexibility when implementing a Dataset class. You are required to implement three methods and you can optionally add other methods depending on your source data The

required methods are __init__(), __len__(), and __getitem__(). The demo PeopleDataset defines its __init__() method as:

```
def __init__(self, src_file, num_rows=None):
  x_tmp = np.loadtxt(src_file, max_rows=num_rows,
    usecols=range(0,7), delimiter="\t",
    skiprows=0, dtype=np.float32)
  y_tmp = np.loadtxt(src_file, max_rows=num_rows,
    usecols=7, delimiter="\t", skiprows=0,
    dtype=np.long)

  self.x_data = T.tensor(x_tmp,
    dtype=T.float32).to(device)
  self.y_data = T.tensor(y_tmp,
    dtype=T.long).to(device)
```

The __init__() method loads data into memory from file using the NumPy loadtxt() function and then converts the data to PyTorch tensors. Instead of using loadtxt(), two other common approaches are to use a program-defined data loading function, or to use the read_csv() function from the Pandas code library. The max_rows parameter of loadtxt() can be used to limit the amount of data read. If max_rows is set to None, then all data in the source file will be read.

In situations where your source data is too large to fit into memory, you will have to read data into a buffer and refill the buffer when the buffer has been emptied. This is a fairly difficult task.

The demo data stores both the predictor values and the dependent values-to-predict in the same file. In situations where the predictor values and dependent variable values are in separate files, you'd have to pass in two source file names instead of just one. Another common alternative is to pass in just a single source directory and then use hard-coded file names for the training and test data.

The demo __init__() method bulk-converts all NumPy array data to PyTorch tensors. An alternative is to leave the data in memory as NumPy arrays and then convert to batches of data to tensors in the __getitem__() method. Conversion from NumPy array data to PyTorch tensor data is an expensive operation so it's usually better to convert just once rather than repeatedly converting batches of data.

The __len__() method is defined as:

```
  def __len__(self):
    return len(self.x_data)
```

A Dataset object has to know how much data there is so that an associated DataLoader object knows how to iterate through all data in batches.

The __getitem__() method is defined as:

```
  def __getitem__(self, idx):
    if T.is_tensor(idx):
      idx = idx.tolist()
    preds = self.x_data[idx, 0:7]
    pol = self.y_data[idx]
    sample = \
      { 'predictors' : preds, 'political' : pol }
    return sample
```


It's common practice to name the parameter which specifies which data to fetch as "idx" but this is somewhat misleading because the idx parameter is usually a Python list of several indexes. The __getitem__() method checks to see if the idx parameter is a PyTorch tensor instead of a Python list, and if so, converts the tensor to a list. The method return value, sample, is a Python Dictionary object and so you must specify names for the dictionary keys ("predictors" in the demo) and the dictionary values ("political" in the demo).

Using a Dataset in a DataLoader

The demo program creates a relatively simple DataLoader object using just the Dataset object plus the batch_size and shuffle parameters:

```
  train_file = ".\\people_train.txt"
  train_ds = PeopleDataset(train_file, num_rows=8)

  bat_size = 3
  train_ldr = T.utils.data.DataLoader(train_ds,
    batch_size=bat_size, shuffle=True)
```

The other eight DataLoader parameters are not used very often. These parameters and their default values are presented in the table in **Figure 2**.

parameter name	default value	brief description
sampler	None	allows a custom data loading sequence
batch_sampler	None	allows a custom sequencing for a batch
num_workers	0	allows sub-processes to load data
collate_fn	None	function to collate samples into a batch
pin_memory	False	pins memory for faster GPU access
drop_last	False	drops the last batch if its size is not the same as all other batches
timeout	0	max time allowed for sub-processes if num_workers > 0
worker_init_fn	None	custom function called by worker processes

[Click on image for larger view.]

Figure 2: PyTorch DataLoader Rarely Used Optional Parameters

In some situations, instead of using a DataLoader to consume the data in a Dataset, it's useful to iterate through a Dataset directly. For example:

```
def process_ds(model, ds):
  # ds is an iterable Dataset of tensors
  for i in range(len(ds)):
    inpts = ds[i]['predictors']
    trgt = ds[i]['target']
    oupt = model(inpts)
    # do something

  return some_value
```


You can use this pattern to compute model accuracy or a custom error metric.

Using Other DataLoaders

Once you understand how to create a custom Dataset and use it in a DataLoader, many of the built-in PyTorch library Dataset objects make more sense than they might otherwise. For example, the TorchVision module has data and functions that are useful for image processing. One of the Dataset classes in TorchVision holds the MNIST image data. There are 70,000 MNIST images. Each image is a handwritten digit from '0' to '9'. Each image has size 28 x 28 pixels and pixels are grayscale values from 0 to 255.

A Dataset class for the MNIST images is defined in the torchvision.datasets package and is named MNIST. You can create a Dataset for MNIST training images like so:

```
import torchvision as tv
tform = tv.transforms.Compose([tv.transforms.ToTensor()])

mnist_train_ds = tv.datasets.MNIST(root=".\\MNIST_Data",
  train=True, transform=tform, target_transform=None,
  download=True)
```

"Passwords are a hassle to use, and they present security risks for users and organizations of all sizes."

'Is WPF Dead?' Some Devs Claim 'Yes' as Microsoft Relegates Issues/PRs to the Community

Microsoft: "We now switch to the model where we accept a lot of PRs from the community because we think of WPF as very mature project so not that much rapid development is happening in WPF area, but we totally support it."

Microsoft Trumpets 2 Million Java Devs on VS Code

The team maintains extensions, including the Extension Pack for Java (16.4 million installs), which bundles six individual extensions that provide the VS Code Java experience.

MOST POPULAR

Please type the letters/numbers you see above.

SUBMIT

Most Popular Articles

'Is WPF Dead?' Some Devs Claim 'Yes' as Microsoft Relegates Issues/PRs to the Community

The Traveling Salesman Problem Using an Evolutionary Algorithm with C#

Spring Cloud Azure 4.5 Furthers Microsoft's 'Passwordless' Push

VS Code Preview: Python in the Browser, Executed by WebAssembly

Did .NET MAUI Ship Too Soon? Devs Sound Off on 'Massive Mistake'

MOST POPULAR

Free Webcasts

Build vs Buy: Is Managing Customer Identity Slowing Your Time to Market?

Is it ODD to shift left? Building elite DevSecOps performers

MARA: The Journey to Open-Telemetry

Video: SolarWinds Observability - A Unified Full Stack Solution for DevOps

> More Webcasts

Upcoming Events

VSLive! 2-Day Hands-On Training Seminar: Learn to Use the Web API in .NET 6/7
February 27-28, 2023 [Virtual]

VSLive! Las Vegas
March 19-24, 2023

VSLive! 2-Day Hands-On Training Seminar: Design, Build and Deliver a Microservices Solution the Cloud Native Way
April 13-14, 2023 [Virtual]

VSLive! Nashville
May 15-19, 2023

VSLive! Microsoft HQ
July 17-21, 2023