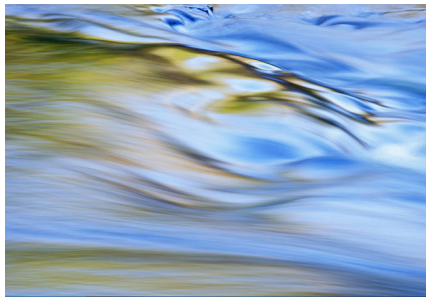
THE DATA SCIENCE LAB

How To: Create a Streaming Data Loader for PyTorch

When training data won't fit into machine memory, a streaming data loader using an internal memory buffer can help. Dr. James McCaffrey shows how, with full code samples.

By James McCaffrey 04/01/2021

GET CODE DOWNLOAD



When using the PyTorch neural network library to create a machine learning prediction model, you must prepare the training data and write code to serve up the data in batches. In situations where the training data is too large to fit into machine memory, one approach is to write a data loader that streams the data using an internal memory buffer. This article shows you how to create a streaming data loader for large training data files.

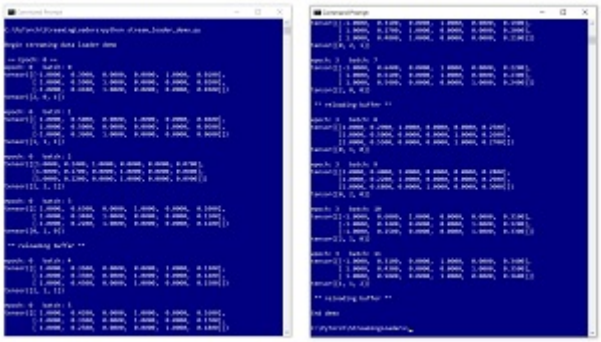
MOST POPULAR

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 demo program uses a dummy data file with just 40 items. The source data is tab-delimited and looks like:

```
-1  0.39  0  0  1  0.0100  2
 1  0.59  1  0  0  0.0200  0
-1  0.41  1  0  0  0.0300  1
 . . .
-1  0.43  0  1  0  0.4000  0
```

You can imagine that each line of data represents a person, and the fields are sex (male = -1, female = +1), age, office location (one of three cities), income, and job type (0, 1, or 2). The age values have been normalized by dividing by 100, the office locations have been one-hot encoded, the incomes have been artificially set to 0.01 to 0.40 so that it's easy to track data

items, and the job type is ordinal encoded as 0, 1, or 2. The hypothetical goal is a multi-class classification model that predicts job type from the other four fields.



[Click on image for larger view.]

Figure 1: A Streaming Data Loader in Action

The streaming data loader sets up an internal buffer of 12 lines of data, a batch size of 3 items, and sets a shuffle parameter to False so that the 40 data items will be processed in sequential order. The demo program instructs the data loader to iterate for four epochs, where an epoch is one pass through the training data file.

After the first four batches of three items have been delivered, the internal buffer has been used up and so the internal buffer is reloaded with the next 12 lines of data. After 36 lines of data have been processed (three buffers worth), there are only four items left, which is not enough to refill the buffer and so the streaming data loader resets to the beginning of the file to start a second epoch.

This article assumes you have an intermediate or better familiarity with a C-family programming language, preferably Python, but doesn't assume you know very much about PyTorch. The complete source code for the demo program, and the dummy data, are presented in this article. The code and data are also available in the accompanying file download.

To run the demo program, you must have Python and PyTorch installed on your machine. The demo programs were developed on Windows 10 using the Anaconda 2020.02 64-bit distribution (which contains Python 3.7.6) and PyTorch version 1.7.0 for CPU installed via pip. You can find detailed step-by-step installation instructions for this configuration in my **blog post**.

When Are Streaming Data Loaders Needed?

In situations where all of the training data will fit into machine memory, the most common approach is to define a problem-specific Dataset class and use a built-in DataLoader object. For the dummy employee data, a Dataset could be defined as shown in **Listing 1**.

Listing 1: Dataset and DataLoader for Non-Huge Training Data

```
    all_xy = np.loadtxt(src_file, max_rows=n_rows,
      usecols=range(0,7), delimiter="\t",
      comments="#", dtype=np.float32)
    tmp_x = all_xy[:, 0:6]  # predictors
    tmp_y = all_xy[: ,6]    # job type

    self.x_data = T.tensor(tmp_x, dtype=T.float32).to(device)
    self.y_data = T.tensor(tmp_y, dtype=T.int64).to(device)

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

  def __getitem__(self, idx):
    preds = self.x_data[idx]
    trgts = self.y_data[idx]
    return (preds, trgts)  # a tuple
```


The __init__() method uses the loadtxt() function to read the entire data file into memory as a NumPy two-dimensional array, and then converts the data into PyTorch tensors. The __getitem__() method returns a single tuple that holds six predictor values and the job type target.

With an employee Dataset class defined, data can be served up in batches for training with code like this:

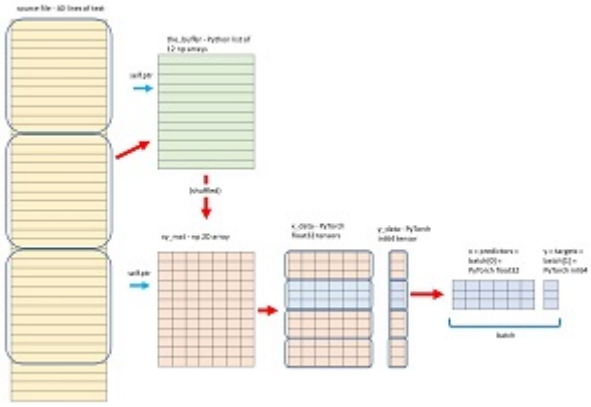
```
train_file = ".\\Data\\employee_train_40.txt"
train_ds = EmployeeDataset(train_file)
train_ldr = T.utils.data.DataLoader(train_ds,
  batch_size=3, shuffle=True)
for epoch in range(0, 4):
  for (batch_idx, batch) in enumerate(train_ldr):
    X = batch[0]     # inputs (sex, age, etc.)
    Y = batch[1]     # targets (job type)
. . .
```

The __getitem__() method is called behind the scenes to fetch the next batch of data. This approach is simple and effective, but it only works if the entire training data file can be stored into memory. If training data is too large to fit into memory, a crude approach is to physically divide the training data into smaller files. This approach is quite common but is messy to implement and difficult to manage. In many situations with very large training data files a better approach is to write a streaming data loader that reads data into a memory buffer, serves data from the buffer, reloading the buffer from file when needed.

Note that in addition to the Dataset class, PyTorch has an IterableDataset class. However, when an IterableDataset object is fed to a DataLoader object, the shuffle parameter is not available. This makes IterableDataset unsuited for training data.

A Streaming Data Loader

The design of the streaming data loader is shown in the diagram in **Figure 2**. The code for the streaming data loader for the dummy employee data file is presented in **Listing 2**. I'll walk through the code, explaining which parts are boilerplate and which parts should be modified for different sets of data.



[Click on image for larger view.]

Figure 2: Streaming Data Loader Design

The streaming data loader design assumes that the buffer size is evenly divisible by the batch size. However, the design does not assume that the file size (number of lines) is evenly divisible by the buffer size. This means that the last few lines of the training data file could be excluded. This design is called drop-last. You can avoid dropping the last few lines of data by setting the batch size and file size so that the file size is evenly divisible by the buffer size, but this is not always feasible.

Listing 2: A Streaming Data Loader Example

```
class EmployeeStreamLoader():
```

```
  def __init__(self, fn, bat_size, buff_size, \
    shuffle=False, seed=0):
    if buff_size % bat_size != 0:
      raise Exception("buff_size must be evenly div by bat_size")

    self.bat_size = bat_size
    self.buff_size = buff_size
    self.shuffle = shuffle

    self.rnd = np.random.RandomState(seed)

    self.ptr = 0              # points into x_data and y_data
    self.fin = open(fn, "r")  # line-based text file
```

The streaming data loader is encapsulated as a class rather than standalone functions. The overall structure of the class is:

```
class EmployeeStreamLoader():
  def __init__(self, fn, bat_size, buff_size, \
    shuffle=False, seed=0): . . .
  def reload_buffer(self): . . .
  def __iter__(self): . . .
  def __next__(self): . . .
```


The class implements __iter__() and __next__() methods so that an EmployeeStreamLoader object is iterable. The __init__() method accepts parameters fn (filename) that points to the source data, bat_size (batch size), buff_size (buffer size), shuffle to control if the data is served up in order for testing or in scrambled order for training, and seed to control the shuffle randomization.

The __init__() method implementation begins with:

```
def __init__(self, fn, bat_size, buff_size, \
  shuffle=False, seed=0):
  if buff_size % bat_size != 0:
    raise Exception("buff must be div by bat_size")
```

```
    self.bat_size = bat_size
    self.buff_size = buff_size
    self.shuffle = shuffle
    self.rnd = np.random.RandomState(seed)
  . . .
```

The class defines a local NumPy RandomState object for use when shuffling the buffer into scrambled order. This approach is generally preferable to relying on the global NumPy random class, so that results don't have a dependency on other calls to random functions.

The __init__() method code concludes with:

```
self.ptr = 0              # points into x_data and y_data
self.fin = open(fn, "r")  # line-based text file
self.the_buffer = []      # list of numpy vectors
self.xy_mat = None        # NumPy 2-D version of buffer
self.x_data = None        # predictors as Tensors
self.y_data = None        # targets as Tensors
self.reload_buffer()
```


The core buffer object is a Python list of NumPy vectors. Each vector represents one line of data. The xy_mat is a NumPy two-dimensional version of the buffer. The x_data and y_data are PyTorch tensors that hold the predictors and the targets. In most scenarios you will not need to modify the definition of the __init__() method. One exception is when you are working with an autoencoder architecture where there are no explicit target values.

Fetching a Batch from the Buffer

The __next__() method serves up a batch of training data. In pseudo-code, the algorithm is:

```
if buffer is empty then
  reload the buffer from file

if the buffer is ready then
  fetch a batch from buffer and return it

if buffer not ready, reached EOF so
  reload buffer for next pass through file
  signal no next batch using StopIteration
```

The definition of the __next__() method begins with these statements:

```
def __next__(self):  # next batch as a tuple
  res = 0
  if self.ptr + self.bat_size > self.buff_size:  # reload
    print(" ** reloading buffer ** ")
    res = self.reload_buffer()
    # 0 = success, -1 = hit eof, -2 = not fully loaded
```

The code checks to see if the buffer needs reloading. Suppose the buff_size is 12. The last index of the buffer is [11]. And suppose the ptr is 9 and the bat_size is 3, then 9 + 3 = 12 which is in range because the start and end indices are inclusive, exclusive -- buffer[9:12] will access rows 9 to 11 inclusive. The indexing in many buffering systems is very tricky and error-prone.

If the buffer needs to be reloaded, method reload_buffer() is called. The return values are 0 for a successful reload, -1 to signal end-of-file was hit when trying to reload, and -2 if only a partial buffer was filled (this indicates a logic error if the buffer size is evenly divisible by the batch size). If the reload was successful, a batch of data is extracted from the tensor version of the buffer and returned as a tuple:

Featured

MOST POPULAR

Report: Mac Developers Prefer VS Code over Xcode

About 57 percent of respondents chose VS Code, followed by Xcode (34 percent), Sublime Text (15 percent), IntelliJ (13 percent), PhpStorm (9 percent) and Nova (5 percent).

Survey Reveals Bigger C# Community, Most and Least Popular Uses

Polling more than 19,000 developers, the new "Developer Economics State of the Developer Nation, 20th Edition," report is out, finding that C# has ticked up a notch in popularity, overtaking PHP for No. 5 on that ranking. What's more, the big twice-yearly report identifies what areas are most and least popular for coding in Microsoft's flagship programming language.

'Epic Fail': ASP.NET PM Struggles with Blazor Hot Reload in Live Demo

THE DATA SCIENCE LAB

How To: Create a Streaming Data Loader for PyTorch

04/01/2021

GET CODE DOWNLOAD

```
if res == 0:
  start = self.ptr
  end = self.ptr + self.bat_size
  x = x_data[start:end, :]
  y = y_data[start:end]
  self.ptr += self.bat_size
  return (x,y)
```


If the buffer reload was not successful, that means end-of-file was reached. The code prepares for the next pass through the data file by reloading the buffer and raising the built-in Python StopIteration exception:

```
# reached end-of-epoch (EOF), so signal no more
self.reload_buffer()  # prepare for next epoch
raise StopIteration
```

This signals the calling code that there is no next item. Any iterable calling code on the EmployeeStreamLoader object, such as a for-loop, will terminate and a new training epoch will begin.

Reloading the Buffer

Method reload_buffer() does most of the work of the streaming data loader, and the method has most of the customization points. The method definition begins:

```
def reload_buffer(self):
  self.the_buffer = []
  self.ptr = 0
  ct = 0       # number of lines read
```

The ptr variable keeps track of where the next batch in the buffer is located. The ct variable tracks how many lines of the source file have been read and stored into the buffer. Next, the method attempts to load buff_size lines of data:

```
while ct < self.buff_size:
  line = self.fin.readline()
  if line == "":
    self.fin.seek(0)
    return -1  # reached EOF
  else:
    line = line.strip()  # remove trailing newline
    np_vec = np.fromstring(line, sep="\t")
    self.the_buffer.append(np_vec)
    ct += 1
```


In the demo code, the line separator is hard-coded as a tab character. You must change this if your data is separated by commas or whitespace.

Python returns an empty string rather than an end-of-file marker when there are no more lines of data to read from a text file. If end-of-file is reached, the file pointer is reset to the beginning of the file. You don't want to close the file because there may be several passes through the file, one pass per training epoch. If a line has been read from file, it is converted to a NumPy vector using the handy fromstring() function, and then the vector is appended to the buffer.

After the while-loop terminates, the method checks to make sure the buffer was fully loaded. Then the method scrambles the buffer into random order if the shuffle parameter is set to True:

```
if len(self.the_buffer) != self.buff_size:
  return -2  # buffer was not fully loaded
if self.shuffle == True:
  self.rnd.shuffle(self.the_buffer)  # in-place
```

The method concludes by converting the list of NumPy vectors to a NumPy array. The predictors (sex, age, city, income) are extracted from columns [0] to [5] inclusive as float32 values, and the class labels (job type) are extracted from column [6] as type int64 (required for PyTorch multi-class classification). The extracted arrays are converted to PyTorch tensors.

```
self.xy_mat = np.array(self.the_buffer)    # 2-D array
self.x_data = T.tensor(self.xy_mat[:, 0:6], \
  dtype=T.float32).to(device)
self.y_data = T.tensor(self.xy_mat[:, 6], \
  dtype=T.int64).to(device)
return 0  # buffer successfully loaded
```

This block of code is where almost all of your customization will take place. In some situations you will want to perform normalization of numeric values. The 40-item dummy employee data already has normalized age values and normalized income values. But suppose the age values in column [1] were in raw format such as 31, 28, 44, and so on. You could normalize age values on the fly by dividing column [1] of the xy_mat array by 100.

Using the Streaming Loader

The demo program begins execution with:


```
def main():
  print("Begin streaming data loader demo ")
  np.random.seed(1)
  fn = ".\\Data\\employee_train_40.txt"  # 40 lines of data
  bat_size = 3
  buff_size = 12  # a multiple of bat_size
  emp_ldr = EmployeeStreamLoader(fn, bat_size, buff_size, \
    shuffle=False)
```

Because the EmployeeStreamLoader has its own internal random number generator, it's not really necessary to set the NumPy random seed, but it's good practice to do so because other components of a PyTorch program will likely use NumPy random numbers. A stream loader object is instantiated as you'd expect. The default parameter value of 1 is used for the object's internal random number generator seed.

The streaming data loader is called in a for-loop with the Python enumerate() function:

```
max_epochs = 4
for epoch in range(max_epochs):
  print(" == Epoch: " + str(epoch) + " == ")
  for (b_idx, batch) in enumerate(emp_ldr):
    print("epoch: " + str(epoch) + "   batch: " + str(b_idx))
    print(batch[0])  # predictors
    print(batch[1])  # labels
    print("")
emp_ldr.fin.close()
print("End demo ")
```

The enumerate() function tracks a built-in counter variable, but this is optional. The for-loop could have been written without enumerate() as:

```
for batch in emp_ldr:
  . . .
```

After all data has been processed, the demo program explicitly closes the streaming loader file pointer. Recall that the EmployeeStreamLoader object opens the source file for reading, and keeps the file open because the object doesn't know how many times the file will be traversed.

The Dummy Data File

For the sake of completeness, the 40-item dummy employee data is listed below, with tab characters replaced by three consecutive blank spaces. If you scrape the data, you may have to replace whitespace with tab character separators.

```
-1   0.39   0   0   1   0.01   2
 1   0.59   1   0   0   0.02   0
-1   0.41   1   0   0   0.03   1
 1   0.50   0   1   0   0.04   1
 1   0.50   0   0   1   0.05   1
-1   0.36   1   0   0   0.06   1
 1   0.24   1   0   0   0.07   2
 1   0.27   0   1   0   0.08   2
 1   0.22   0   1   0   0.09   1
 1   0.63   0   1   0   0.10   0
 1   0.34   1   0   0   0.11   1
 1   0.23   1   0   0   0.12   0
```

```
-1   0.22   1   0   0   0.12   0
 1   0.35   0   0   1   0.13   2
-1   0.33   0   1   0   0.14   1
 1   0.45   0   1   0   0.15   1
 1   0.42   0   1   0   0.16   1
```

Wrapping Up

There are some indications that current brute force approaches for training machine learning systems with huge data files is becoming unsustainable. It's estimated that training the GPT-3 language model cost approximately $4.6 million dollars of processing time. And it's not unusual for the computing cost of training even a moderately sized machine learning model to exceed $10,000. There are many research efforts under way to find ways to train machine learning models more efficiently with smaller datasets.

« previous 1 | 2

About the Author

Dr. James McCaffrey works for Microsoft Research in Redmond, Wash. He has worked on several Microsoft products including Azure and Bing. James can be reached at jamccaff@microsoft.com.

MOST POPULAR

PRINTABLE FORMAT

ALSO ON VISUAL STUDIO MAGAZINE

Microsoft Details 9 Desktop Dev …
a month ago • 8 comments
For .NET coders targeting Windows, the choices boil down to more traditional …

.NET 6 Preview 3 Furthers 'Hot Reload …
19 days ago • 3 comments
.NET 6 Preview 3 includes early Hot Reload support for ASP.NET Core/Blazor …

Microsoft's Lander on Blazor Desktop: 'I …
2 months ago • 17 comments
For all of the talk of unifying the disparate ecosystem of Microsoft-centric …

Proj Prev
2 mor
In an point every