batchup Documentation

Release 0.2.2

Geoff French

Jul 05, 2019

Contents

BatchUp is a lightweight Python library for extracting mini-batches of data for the purpose of training neural networks.

A quick example:

```
from batchup import data_source

# Construct an array data source
ds = data_source.ArrayDataSource([train_X, train_y])

# Iterate over samples, drawing batches of 64 elements in
# random order
for (batch_X, batch_y) in ds.batch_iterator(batch_size=64,
                                            shuffle=True):
    # Processes batches here...
```

CHAPTER 1

User Guide

1.1 Installation

You can install BatchUp from PyPI with:

```
> pip install batchup
```

1.1.1 Bleeding edge version From GitHub

Alternatively, clone the GitHub repository at http://github.com/Britefury/batchup then from within the newly cloned repo:

```
> python setup.py develop
```

1.2 Basic batch iteration from arrays

BatchUp defines *data sources* from which we draw data in mini-batches. They are defined in the `data_source` module and the one that you will most frequency use is `data_source.ArrayDataSource`.

Contents

1.2.1 Simple batch iteration from NumPy arrays

This example will show how to draw mini-batches of samples from NumPy arrays in random order.

Assume we have a training set in the form of NumPy arrays in the variables `train_X` and `train_y`.

First, construct a *data source* that will draw data from `train_X` and `train_y`:

```
from batchup import data_source

# Construct an array data source
ds = data_source.ArrayDataSource([train_X, train_y])
```

`ArrayDataSource` notes:

- `train_X` and `train_y` must have the same number of samples
- you can use any number of arrays when building the `ArrayDataSource`

Now we can use the `batch_iterator()` method to create a batch iterator from which we can draw mini-batches of data:

```
# Iterate over samples, drawing batches of 64 elements in
# random order
for (batch_X, batch_y) in ds.batch_iterator(
        batch_size=64, shuffle=np.random.RandomState(12345)):
    # Processes batch_X and batch_y here...
```

Batch iterator notes:

- the last batch will be short (have less samples than the requested batch size) if there isn't enough data to fill it
- **the `shuffle` parameter:**
  - using `shuffle=True` will use NumPy's default random number generator
  - if no value is provided for `shuffle`, samples will be processed in-order

Note: we don't *have* to use NumPy arrays; any array-like object will do; see *Data from array-like objects (data accessors)* for more.

1.2.2 Iterating over a subset of the samples

We can specify the indices of a subset of the samples in a dataset and draw mini-batches from only those samples:

```
import numpy as np

# Randomly choose a subset of 20,000 samples, by indices
subset_a = np.random.permutation(len(train_X))[:20000]

# Construct an array data source that will only draw samples whose indices are in␣
↪`subset_a`
ds = data_source.ArrayDataSource([train_X, train_y], indices=subset_a)

# Drawing batches of 64 elements in random order
for (batch_X, batch_y) in ds.batch_iterator(
        batch_size=64, shuffle=np.random.RandomState(12345)):
    # Processes batches here...
```

1.2.3 Getting the indices of sample in the mini-batches

We can ask to be provided with the indices of the samples that were drawn to form the mini-batch:

```
# Construct an array data source that will provide sample indices
ds = data_source.ArrayDataSource([train_X, train_y], include_indices=True)

# Drawing batches of 64 elements in random order
for (batch_ndx, batch_X, batch_y) in ds.batch_iterator(
        batch_size=64, shuffle=np.random.RandomState(12345)):
    # Processes batches here; indices in batch_ndx
```

1.2.4 Batches from repeated/looped arrays

Lets say you need an iterator that extracts samples from your dataset and starts from the beginning when it reaches the end. Provide a value for the `repeats` argument of the `ArrayDataSource` constructor like so:

```
ds_times_5 = data_source.ArrayDataSource([train_X, train_y], repeats=5)
```

Now use the `batch_iterator()` method as before.

The `repeats` parameter accepts either `-1` for infinite, or any positive integer `>= 1` for a specified number of repetitions:

```
inf_ds = data_source.ArrayDataSource([train_X, train_y], repeats=-1)
```

This will also work if the dataset has less samples than the batch size; this is not a common use case but it can happen.

1.3 Data augmentation on-the-fly

We can define a function that applies data augmentation on the fly. Let's assume that the images batches that we draw have the shape `(sample, channel, height, width)` and that we wish to randomly choose 50% of the images in the batch to be horizontally flipped:

```
# Define our batch augmentation function.
def augment_batch(batch_X, batch_y):
    # Create an array of random 0's and 1's with 50% probability
    flip_flags = np.random.binomial(1, 0.5, size=(len(batch_X),))

    # Convert to `bool` dtype.
    flip_flags = flip_flags.astype(bool)

    # Flip the horizontal dimension in samples identified by
    # `flip_flags`
    batch_X[flip_flags, ...] = flip_flags[flip_flags, :, :, ::-1]

    # Return the batch as a tuple
    return batch_X, batch_y
```

We can add our `augment_batch` function to our batch extraction pipeline by invoking the `map()` method like so:

```
# Construct an array data source from ``train_X`` and ``train_y``
ds = data_source.ArrayDataSource([train_X, train_y])
```

(continues on next page)

(continued from previous page)

```
# Apply augmentation using `augment_batch`
ds = ds.map(augment_batch)

# Drawing batches of 64 elements in random order
for (batch_X, batch_y) in ds.batch_iterator(
        batch_size=64, shuffle=np.random.RandomState(12345)):
    # Processes batches here...
```

1.4 Loading images from disk on-the-fly: data from array-like objects

So far we've only seen how to extract batches from NumPy arrays. This is not always possible or convenient. For example we may wish to load images from files on disk on-the-fly, as loading all of the images into memory may exhaust the available RAM. We will now demonstrate how load images on-the-fly from files on-disk.

Contents

1.4.1 Data from array-like objects (data accessors)

`data_source.ArrayDataSource` is quite flexible; it can draw data from array-like objects, not just NumPy arrays. This allows us to customise how the data is acquired.

Our array-like object must define the following:

- a `__len__` method that returns the number of available samples
- a `__getitem__` method that retrieves samples identified by index; the index can either be an integer or a NumPy array of indices that identify multiple samples to retrieve in one go

Lets define an array-like image accessor that is given a list of paths that identify images to load. We will define a helper method that will use Pillow to load the image:

```
import six
import numpy as np
from PIL import Image

class ImageFileAccessor (object):
    def __init__(self, paths):
        # Paths is a list of file paths
        self.paths = paths

    # We have to imlement the `__len__` method:
    def __len__(self):
        return len(self.paths)

    # We have to implement the `__getitem__` method that
```

(continues on next page)

(continued from previous page)

```
    # `ArrayDataSource` will use to get samples
    def __getitem__(self, index):
        # Check if its a built-in Python integer *or* a NumPy
        # integer type
        if isinstance(index, six.integer_types + (np.integer,)):
            return self._load_image(self.paths[index])
        elif isinstance(index, np.ndarray):
            if index.ndim != 1:
                raise ValueError(
                    'index array should only have 1 dimension, '
                    'not {}'.format(index.ndim))
            images = [self._load_image(self.paths[i]) for i in index]
            return np.stack(images, axis=0)
        else:
            raise TypeError('index should be an integer or a NumPy '
                            'array, not a {}'.format(type(index))

    # Define a helper method for loading an image
    def _load_image(self, path):
        # Load the image using Pillow and convert to a NumPy array
        img = Image.open(path)
        return np.array(img)
```

We can now gather the paths of some image files and pass them to `ImageFileAccessor` and use it in place of a NumPy array:

```
import glob
import os

# Put your dataset path here
PATH = '/path/to/images'

# Get image paths
image_paths = glob.glob(os.path.join(PATH, '*.jpg'))

# Build our array-like image file accessor
train_X = ImageFileAccessor(image_paths)

# Let's assume `train_y` is a NumPy array

# Mixing custom array types with NumPy arrays is fine
ds = data_source.ArrayDataSource([train_X, train_y])

for (batch_X, batch_y) in ds.batch_iterator(
        batch_size=64, shuffle=np.random.RandomState(12345)):
    # Process batches here...
```

1.4.2 Lists instead of arrays

The above code will work fine if all the image have the same resolution. If they are of varying sizes `np.stack` will fail.

Also `__getitem__` doesn't have to return NumPy arrays; it can return a single PIL image or a list of PIL images:

```
import six
import numpy as np
from PIL import Image

class NonUniformImageFileAccessor (object):
    def __init__(self, paths):
        # Paths is a list of file paths
        self.paths = paths

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

    def __getitem__(self, index):
        if isinstance(index, six.integer_types + (np.integer,)):
            return self._load_image(self.paths[index])
        elif isinstance(index, np.ndarray):
            if index.ndim != 1:
                raise ValueError('index array should only have 1 dimension, '
                    'not {}'.format(index.ndim))
            return [self._load_image(self.paths[i])
                    for i in index]
        else:
            raise TypeError('index should be an integer or a NumPy '
                            'array, not a {}'.format(type(index))

    # Define a helper method for loading an image
    def _load_image(self, path):
        # Load the image using Pillow
        return Image.open(path)
```

Lets load the Kaggle Dogs vs Cats training set. Also, lets define a batch augmentation function (see *Data augmentation on-the-fly*) that will scale each image to `64 x 64` pixels and convert it to a NumPy array:

```
def augment_batch(batch_X, batch_y):
    out_X = []
    # For each PIL Image in `batch_X`:
    for img in batch_X:
        # Resize to 64 x 64
        img = img.resize((64, 64))

        # PIL allows you to easily get the image data as
        # a NumPy array
        x = np.array(img)

        out_X.append(x)

    # Stack the images into one array
    out_X = np.stack(out_X, axis=0)

    return (out_X, batch_y)


# Put your dataset path here
PATH = '/path/to/dogs_vs_cats'

# Get paths to the training set images
image_paths = glob.glob(os.path.join(PATH, 'train', '*.jpg'))
```

(continues on next page)

(continued from previous page)

```
# Build our array-like image file accessor
train_X = NonUniformImageFileAccessor(image_paths)

# Construct our ground truths as a NumPy array
# The ground truth class is determined by the prefix
train_y = [(1 if os.path.basename(p).startswith('dog') else 0)
           for p in image_paths)]
train_y = np.array(train_y, dtype=np.int32)

# Mixing custom array types with NumPy arrays is fine
kaggle_ds = data_source.ArrayDataSource([train_X, train_y])

# Apply augmentation function
kaggle_ds = kaggle_ds.map(augment_batch)

for (batch_X, batch_y) in kaggle_ds.batch_iterator(
        batch_size=64, shuffle=np.random.RandomState(12345)):
    # Process batches here...
```

1.4.3 Performance issues

Loading images from disk in this way can incur a significant performance overhead due to disk access and decompressing the image data once it has been loaded into memory. It would be desirable to do this in a separate thread or process in order to hide this latency. You can learn how to do this in *Parallel processing for faster batching*.

1.5 Parallel processing for faster batching

Batch generation pipelines and involve reading images from disk on the fly and applying potentially expensive data augmentation operations can incur significant overhead. Running such a pipeline in a single threaded/process along side the training code can result in the GPU spending a significant amount of time waiting for the CPU to provide it with data.

BatchUp provides worker pools that run the batch generation pipeline in background processes or threads, ameliorating this problem.

Contents

1.5.1 Using threads

Let's build on the *Data from array-like objects (data accessors)* example in which we load the Kaggle Dogs vs Cats training set. Let's assume that `train_X` references an instance of the `NonUniformImageFileAccessor` class defined in that example, `train_y` contains ground truth classes and that we have the augmentation function `augment_batch`:

```
from batchup import data_source, work_pool

# Build a pool of 4 worker threads:
th_pool = work_pool.WorkerThreadPool(processes=4)

# Build our data source and apply augmentation function
ds = data_source.ArrayDataSource([train_X, train_y])

ds = ds.map(augment_batch)

# Construct a parallel data source that prepares mini-batches in the
# background. It wraps the existing data source `ds` and will try
# to keep a buffer of 32 mini-batches full to eliminate latency:
par_ds = th_pool.parallel_data_source(ds, batch_buffer_size=32)

# As soon as we create an iterator, it will start filling its
# buffer; lets create an iterator right now to get it going in
# the background:
par_iter = par_ds.batch_iterator(
    batch_size=64, shuffle=np.random.RandomState(12345))

# Do some other initialisation stuff that may take a while...

# By now, with any luck, some batches will be ready to retrieve

for (batch_X, batch_y) in par_iter:
    # Process batches here...
```

1.5.2 Using processes

In some cases the data source that you wish to parallelize may not be thread safe or may perform poorly in a multi-threaded environment due to Pythons' Global Interpreter Lock (GIL). In such cases you can use process based pools that use separate processes rather than threads.

There are two issues that you should be aware of when using processes:

- Process-based pools incur a higher overhead due to having to copy batches between processes. We use `MemmappingPool` from `joblib` to attempt to ameliorate this, but that mechanism only works with NumPy arrays, so some pickling will still be performed
- Any classes or functions used – that includes custom accessor classes and data augmentation functions – must be declared in the top level of a module so that pickle can find them for the purpose of passing references to them to the worker processes

That aside, we need only replace the reference to `WorkerThreadPool` with `WorkerProcessPool`:

```
# Build a pool of 4 worker processes:
proc_pool = work_pool.WorkerProcessPool(processes=4)

# Construct a data source that prepares mini-batches in the
# background. It wraps the existing data source `ds` and will try
# to keep a buffer of 32 mini-batches full to eliminate latency:
par_ds = proc_pool.parallel_data_source(ds, batch_buffer_size=32)

# ... use `par_ds` the same way as before ...
```

1.6 Iterating over data sets of different sizes (e.g. for semi-supervised learning)

There are some instances where we wish to draw samples from two data sets simultaneously, where the data sets have different sizes. One example is in semi-supervised learning where we have a small dataset of labeled samples `lab_X` with ground truths `lab_y` and a larger set of unlabeled samples `unlab_X`. Lets say we want a single epoch to consist of the entire unlabeled dataset while looping over the labeled dataset repeatly. The `data_source.CompositeDataSource` class can help us here.

Without using `CompositeDataSource` we would need the following:

```
rng = np.random.RandomState(12345)

# Construct the data sources; the labeled data source will
# repeat infinitely
ds_lab = data_source.ArrayDataSource([lab_X, lab_y], repeats=-1)
ds_unlab = data_source.ArrayDataSource([unlab_X])

# Construct an iterator to get samples from our labeled data source:
lab_iter = ds_lab.batch_iterator(batch_size=64, shuffle=rng)

# Iterate over the unlabeled data set in the for-loop
for (batch_unlab_X,) in ds_unlab.batch_iterator(
        batch_size=64, shuffle=rng):
    # Extract batches from the labeled iterator ourselves
    batch_lab_X, batch_lab_y = next(lab_iter)

    # (we could also use `zip`)

    # Process batches here...
```

We can use `CompositeDataSource` to simplify the above code. It will drawn samples from both `ds_lab` and `ds_unlab` and will shuffle the samples from these data source idependently:

```
# Construct the data sources; the labeled data source will
# repeat infinitely
ds_lab = data_source.ArrayDataSource([lab_X, lab_y], repeats=-1)
ds_unlab = data_source.ArrayDataSource([unlab_X])
# Combine with a `CompositeDataSource`
ds = data_source.CompositeDataSource([ds_lab, ds_unlab])

# Iterate over both the labeled and unlabeled samples:
for (batch_lab_X, batch_lab_y, batch_unlab_X) in ds.batch_iterator(
        batch_size=64, shuffle=rng):
    # Process batches here...
```

You can also have `CompositeDataSource` generate structured mini-batches that reflect the structure of the component data sources. The batches will have a nested tuple structure:

```
# Disable flattening with `flatten=False`
ds_struct = data_source.CompositeDataSource(
    [ds_lab, ds_unlab], flatten=False)

# Iterate over both the labeled and unlabeled samples.
for ((batch_lab_X, batch_lab_y), (batch_unlab_X,)) in \
        ds_struct.batch_iterator(batch_size=64, shuffle=rng):
    # Process batches here...
```

1.7 Sample weighting to alter likelihood of samples

BatchUp defines *samplers* that are used to generate the indices of samples that should be combined to form a mini-batch. They are defined in the `sampling` module.

When constructing a data source (e.g. `ArrayDataSource`) you can provide a sampler that will control how the samples are selected.

Buy default one of the standard samplers (`StandardSampler` or `SubsetSampler`) will be constructed if you don't provide one.

Contents

1.7.1 The weighted sampler

If you want some samples to be drawn more frequently than others, construct a `WeightedSampler` and pass it as the sampler argument to the py:class:.*ArrayDataSource* constructor. In the example the per-sample weights are stored in `train_w`.

```
from batchup import sampling

sampler = sampling.WeightedSampler(weights=train_w)

ds = data_source.ArrayDataSource([train_X, train_y], sampler=sampler)

# Drawing batches of 64 elements in random order
for (batch_X, batch_y) in ds.batch_iterator(
        batch_size=64, shuffle=np.random.RandomState(12345)):
    # Processes batches here...
```

Note that in-order is NOT supported when using `WeightedSampler`, so `shuffle` cannot be `False` or `None`.

To draw from a subset of the dataset, use `WeightedSubsetSampler`:

```
from batchup import sampling

# NOTE that the weights parameter is called `sub_weights` (rather
# than `weights`) and that it must have the same length as `indices`.
sampler = sampling.WeightedSubsetSampler(sub_weights=train_w[subset_a],
                                         indices=subset_a)

ds = data_source.ArrayDataSource([train_X, train_y], sampler=sampler)

# Drawing batches of 64 elements in random order
for (batch_X, batch_y) in ds.batch_iterator(
        batch_size=64, shuffle=np.random.RandomState(12345)):
    # Processes batches here...
```

1.7.2 Counteracting class imbalance

An alternate constructor method `WeightedSampler.class_balancing_sampler()` is available to construct a weighted sampler to compensate for class imbalance:

```
# Construct the sampler; NOTE that the `n_classes` argument
# is *optional*
sampler = sampling.WeightedSampler.class_balancing_sampler(
    y=train_y, n_classes=train_y.max() + 1)

ds = data_source.ArrayDataSource([train_X, train_y], sampler=sampler)

# Drawing batches of 64 elements in random order
for (batch_X, batch_y) in ds.batch_iterator(
        batch_size=64, shuffle=np.random.RandomState(12345)):
    # Processes batches here...
```

The `WeightedSampler.class_balancing_sample_weights()` helper method constructs an array of sample weights in case you wish to modify the weights first:

```
weights = sampling.WeightedSampler.class_balancing_sample_weights(
    y=train_y, n_classes=train_y.max() + 1)

# Assume `modify_weights` is defined above
weights = modify_weights(weights)

# Construct the sampler and the data source
sampler = sampling.WeightedSampler(weights=weights)
ds = data_source.ArrayDataSource([train_X, train_y], sampler=sampler)

# Drawing batches of 64 elements in random order
for (batch_X, batch_y) in ds.batch_iterator(
        batch_size=64, shuffle=np.random.RandomState(12345)):
    # Processes batches here...
```

CHAPTER 2

API Reference

2.1 batchup.data_source

2.2 batchup.sampling

2.3 batchup.work_pool

CHAPTER 3

Indices and tables

- genindex
- modindex
- search