dataload

```
local dl = require 'dataload'
```

A collection of Torch dataset loaders.
The library provides the following generic data loader classes :

- DataLoader : an abstract class inherited by the following classes;
- TensorLoader : for tensor or nested (i.e. tables of) tensor datasets;
- ImageClass : for image classification datasets stored in a flat folder structure;
- AsyncIterator : decorates a `DataLoader` for asynchronou multi-threaded iteration;
- SequenceLoader : for sequence datasets like language or time-series;
- MultiSequence : for shuffled sets of sequence datasets like shuffled sentences;
- MultiImageSequence : for suffled sets of sequences of input and target images.

The library also provides functions for downloading specific datasets
and preparing them using the above loaders :

- loadMNIST : load the MNIST handwritten digit dataset for image classification;
- loadCIFAR10 : load the CIFAR10 dataset for image classification;
- loadImageNet : load the ILSVRC2014 dataset for image classification;
- loadPTB : load the Penn Tree Bank corpus for language modeling;
- loadGBW : load the Google Billion Words corpus for language modeling;
- loadSentiment140 : load the Twitter data for sentiment analysis/classification (sad, happy).

Also, we try to provide some useful preprocessing functions :

- fitImageNormalize : normalize images by channel.

DataLoader

```
dataloader = dl.DataLoader()
```

An abstract class inherited by all `DataLoader` instances.
It wraps a data set to provide methods for accessing

`inputs` and `targets`. The data itself may be loaded from disk or memory.

[n] size()

Returns the number of samples in the `dataloader`.

[size] isize([excludedim])

Returns the `size` of `inputs`. When `excludedim` is 1 (the default),
the batch dimension is excluded from `size`.
When `inputs` is a tensor, the returned `size` is
a table of numbers. When it is a table of tensors, the returned `size`
is a table of table of numbers.

[size] tsize([excludedim])

Returns the `size` of `targets`. When `excludedim` is 1 (the default),
the batch dimension is excluded from `size`.
When `targets` is a tensor, the returned `size` is
a table of numbers. When it is a table of tensors, the returned `size`
is a table of table of numbers.

[inputs, targets] index(indices, [inputs, targets])

Returns `inputs` and `targets` containing samples indexed by `indices`.

So for example :

```
indices = torch.LongTensor{1,2,3,4,5}
inputs, targets = dataloader:index(indices)
```

would return a batch of `inputs` and `targets` containing samples 1 through 5.
When `inputs` and `targets` are provided as arguments, they are used as
memory buffers for the returned `inputs` and `targets`,
i.e. their allocated memory is reused.

[inputs, targets] sample(batchsize, [inputs, targets])

Returns `inputs` and `targets` containing `batchsize` random samples.
This method is equivalent to :

```
indices = torch.LongTensor(batchsize):random(1,dataloader:size())
inputs, targets = dataloader:index(indices)
```

[inputs, targets] sub(start, stop, [inputs, targets])

Returns `inputs` and `targets` containing `stop-start+1` samples between `start` and `stop`.
This method is equivalent to :

```
indices = torch.LongTensor():range(start, stop)
inputs, targets = dataloader:index(indices)
```

shuffle()

Internally shuffles the `inputs` and `targets`. Note that not all
subclasses support this method.

[ds1, ds2] split(ratio)

Splits the `dataloader` into two new `DataLoader` instances
where `ds1` contains the first `math.floor(ratio x dataloader:size())` samples,
and `ds2` contains the remainder.
Useful for splitting a training set into a new training set and validation set.

[iterator] subiter([batchsize, epochsize, ...])

Returns an iterator over a validation and test sets.
Each iteration returns 3 values :

- `k` : the number of samples processed so far. Each iteration returns a maximum of `batchsize` samples.
- `inputs` : a tensor (or nested table thereof) containing a maximum of `batchsize` inputs.
- `targets` : a tensor (or nested table thereof) containing targets for the commensurate inputs.

The iterator will return batches of `inputs` and `targets` of size at most `batchsize` until `epochsize` samples have been returned.

Note that the default implementation of this iterator is to call sub for each batch.
Sub-classes may over-write this behavior.

Example :

```
local dl = require 'dataload'

inputs, targets = torch.range(1,5), torch.range(1,5)
dataloader = dl.TensorLoader(inputs, targets)

local i = 0
for k, inputs, targets in dataloader:subiter(2,6) do
   i = i + 1
   print(string.format("batch %d, nsampled = %d", i, k))
   print(string.format("inputs:\n%stargets:\n%s", inputs, targets))
end
```

Output :

```
batch 1, nsampled = 2
inputs:
 1
 2
[torch.DoubleTensor of size 2]
targets:
 1
 2
[torch.DoubleTensor of size 2]

batch 2, nsampled = 4
inputs:
```

```
 3
 4
[torch.DoubleTensor of size 2]
targets:
 3
 4
[torch.DoubleTensor of size 2]

batch 3, nsampled = 5
inputs:
 5
[torch.DoubleTensor of size 1]
targets:
 5
[torch.DoubleTensor of size 1]

batch 4, nsampled = 6
inputs:
 1
[torch.DoubleTensor of size 1]
targets:
 1
[torch.DoubleTensor of size 1]
```

Note how the last two batches are of size 1 while those before are of size `batchsize = 2`.
The reason for this is that the `dataloader` only has 5 samples.
So the last batch is split between the last sample and the first.

[iterator] sampleiter([batchsize, epochsize, ...])

Returns an iterator over a training set.
Each iteration returns 3 values :

- `k` : the number of samples processed so far. Each iteration returns a maximum of `batchsize` samples.
- `inputs` : a tensor (or nested table thereof) containing a maximum of `batchsize` inputs.
- `targets` : a tensor (or nested table thereof) containing targets for the commensurate inputs.

The iterator will return batches of `inputs` and `targets` of size at most `batchsize` until `epochsize` samples have been returned.

Note that the default implementation of this iterator is to call sample for each batch.
Sub-classes may over-write this behavior.

Example :

```
local dl = require 'dataload'

inputs, targets = torch.range(1,5), torch.range(1,5)
dataloader = dl.TensorLoader(inputs, targets)

local i = 0
for k, inputs, targets in dataloader:sampleiter(2,6) do
   i = i + 1
   print(string.format("batch %d, nsampled = %d", i, k))
   print(string.format("inputs:\n%stargets:\n%s", inputs, targets))
end
```

Output :

```
batch 1, nsampled = 2
inputs:
 1
 2
[torch.DoubleTensor of size 2]
targets:
 1
 2
[torch.DoubleTensor of size 2]

batch 2, nsampled = 4
inputs:
 4
 2
[torch.DoubleTensor of size 2]
targets:
 4
 2
[torch.DoubleTensor of size 2]

batch 3, nsampled = 6
inputs:
 4
 1
```

```
[torch.DoubleTensor of size 2]
targets:
 4
 1
[torch.DoubleTensor of size 2]
```

reset()

Resets all internal counters such as those used for iterators.
Called by AsyncIterator before serializing the `DataLoader` to threads.

collectgarbage()

Collect garbage every `self.gccdelay` times this method is called.

[copy] clone()

Returns a deep `copy` clone of `self`.

TensorLoader

```
dataloader = dl.TensorLoader(inputs, targets)
```

The `TensorLoader` can be used to encapsulate tensors of `inputs` and `targets`.
As an example, consider a dummy `3 x 8 x 8` image classification dataset consisting of 1000 samples and 10 classes:

```
inputs = torch.randn(1000, 3, 8, 8)
targets = torch.LongTensor(1000):random(1,10)
dataloader = dl.TensorLoader(inputs, targets)
```

The `TensorLoader` can also be used to encapsulate nested tensors of `inputs` and `targets`.
It uses recursive functions to handle nestings of arbitrary depth. As an example, let us modify the above example to include `x,y` GPS coordinates in the `inputs` and a parallel set of classification `targets` (7 classes):

```
inputs = {torch.randn(1000, 3, 8, 8), torch.randn(1000, 2)}
targets = {torch.LongTensor(1000):random(1,10),
torch.LongTensor(1000):random(1,7)}
dataloader = dl.TensorLoader(inputs, targets)
```

ImageClass

```
dataloader = dl.ImageClass(datapath, loadsize, [samplesize,
samplefunc, sortfunc, verbose])
```

For loading an image classification data set stored in a flat folder structure :

```
(datapath)/(classdir)/(imagefile).(jpg|png|etc)
```

So directory `classdir` is expected to contain the all images belonging to that class.
All image files are indexed into an efficient `CharTensor` during initialization.
Images are only loaded into `inputs` and `targets` tensors upon calling batch sampling methods like index, sample and sub.

Note that for asynchronous loading of images (i.e. loading batches of images in different threads),
the `ImageClass` loader can be decorated with an AsyncIterator.
Images on disk can have different height, width and number of channels.

Constructor arguments are as follows :

- `datapath` : one or many paths to directories of images;
- `loadsize` : initialize size to load the images to. Example : `{3, 256, 256}` ;
- `samplesize` : consistent sample size to resize the images to. Defaults to `loadsize` ;
- `samplefunc` : `function f(self, dst, path)` used to create a sample(s) from an image path. Stores them in `CharTensor` `dst` . Strings `"sampleDefault"` (the

default), `"sampleTrain"` or `"sampleTest"` can also be provided as they refer to existing functions
- `verbose` : display verbose message (default is `true`);
- `sortfunc` : comparison operator used for sorting `classdir` to get class indices. Defaults to the `<` operator.

AsyncIterator

```
dataloader = dl.AsyncIterator(dataloader, [nthread, verbose,
serialmode])
```

This `DataLoader` subclass overwrites the `subiter` and `sampleiter` iterator methods. The implementation uses the threads package to build a pool of `nthread` worker threads. The main thread delegates the tasks of building `inputs` and `targets` tensors to the workers. The workers each have a deep copy of the decorated `dataloader`. The optional parameter serialmode can be specified as 'ascii' (default) or 'binary'. If large amounts of data need to be processed, 'binary' can prevent the dataloader from allocating too much RAM.
When a task is received from the main thread through the Queue, they call `sample` or `sub` to build the batch and return the `inputs` and `targets` to the main thread. The iteration is asynchronous as the first iteration will fill the Queue with `nthread` tasks.
Note that when `nthread > 1` the order of tensors is not deterministic.
This loader is well suited for decorating a `dl.ImageClass` instance and other such I/O and CPU bound loaders.

SequenceLoader

```
dataloader = dl.SequenceLoader(sequence, batchsize,
[bidirectional])
```

This `DataLoader` subclass can be used to encapsulate a `sequence` for training time-series or language models.

The `sequence` is a tensor where the first dimension indexes time.
Internally, the loader will split the `sequence` into `batchsize` subsequences.
Calling the `sub(start, stop, inputs, targets)` method will return
`inputs` and `targets` of size `seqlen x batchsize [x inputsize]`
where `stop - start + 1 <= seqlen`.
See RNNLM training script for an example.

The `bidirectional` argument should be set
to `true` for bidirectional models like BRNN/BLSTMs. In which case,
the returned `inputs` and `targets` will be aligned.
For example, using `batchsize = 3` and `seqlen = 5`:

```
print(inputs:t(), targets:t())
   36  1516   853    94  1376
 3193   433   553   805   521
  512   434    57  1029  1962
[torch.IntTensor of size 3x5]

   36  1516   853    94  1376
 3193   433   553   805   521
  512   434    57  1029  1962
[torch.IntTensor of size 3x5]
```

When `bidirectional` is `false` (the default), the `targets` will
be one step in the future with respect to the inputs:
For example, using `batchsize = 3` and `seqlen = 5`:

```
print(inputs:t(), targets:t())
   36  1516   853    94  1376
 3193   433   553   805   521
  512   434    57  1029  1962
[torch.IntTensor of size 3x5]

 1516   853    94  1376   719
  433   553   805   521    27
  434    57  1029  1962    49
[torch.IntTensor of size 3x5]
```

MultiSequence

```
dataloader = dl.MultiSequence(sequences, batchsize)
```

This DataLoader subclass is used by the Billion Words dataset to encapsulate unordered sentences.
The `sequences` arguments is a table or tds.Vec of tensors.
Each such tensors is a single sequence independent of the others. The tensor can be multi-dimensional as long
as the non-sequence dimension sizes are consistent from sequence to sequence.

When calling `sub(start, stop)` or `subiter(seqlen)` methods,
a column of the returned `inputs` and `targets` tensors (of size `seqlen x batchsize`) could
contain multiple sequences. For example, a character-level language model could look like:

```
target : [ ] E L L O [ ] C R E E N ...
input  : [ ] H E L L [ ] S C R E E ...
```

where `HELLO` and `SCREEN` would be two independent sequences.
Note that `[ ]` is a zero mask used to seperate independent sequences.
For most cases, the `[ ]` token is a 0.
Except for 1D `targets`, where it is a 1 (so that it works with `ClassNLLCriterion`).

MultiImageSequence

```
ds = dl.MultiImageSequence(datapath, batchsize, loadsize,
samplesize, [samplefunc, verbose])
```

This `DataLoader` is used to load datasets consisting of independent sequences of
input and target images. So basically, each independent sequence consists of
two sequences of the same size, one for inputs, one for targets.

As a concrete example, this `DataLoader` could be used to
wrap a dataset where each input is a sequence of video frames,
and its commensurate targets are binary masks.

Like the `ImageClass` loader, `MultiImageSequence` expects images to be stored on disk.
Each directory is organized as :

```
[datapath]/[seqid]/[input|target][1,2,3,...,T].jpg
```

where the `datapath` (first constructor argument) specifies the file system path to the data.
That directory is expected to contain a folder for each sequence, here represented by the `seqid` variable.
The `seqid` folder can have any name, but by default its contents are expected to contain the pattern
`input%d.jpg` and `target%d.jpg` for input and target images, respectively.
Internally, the `%d` is replaced with integers starting at `1` until no more images are found.
These patterns can be replaced after construction via the `inputpattern` and `targetpattern`.

Variable length sequences are natively supported.

Images will be only be loaded when requested.

Like the `MultiSequence` loader, the `batchsize` must be specified during construction.
Like the `ImageClass`, the `loadsize` argument specifies that size of to which the images are to be loaded initially.
These are specified as two tables in `c x h x w` format, for inputs and targets respectively (e.g. `{{3,28,28},{1,8,8}}`).

The `samplesize` specifies the returned input image size (e.g. `{3,24,24}`).
The actual sample size of the targets cannot be provided as it will be forced to be proportional to the input's load to sample size.

The `samplefunc` specifies the function to use for sampling input and target images.
The default value of `sampleDefault` simply resizes the images to the given input `samplesize` and the proportional target sample size.
When `sampleTrain` is provided, a random location will be chosen for each sampled sequence.

When calling `sub(start, stop)` the returned `input` and `target` are tensors of size `seqlen x batchsize x samplesize`. Since variable length sequences are natively supported, the returned `inputs` and `targets` will be separated by mask tokens (here represented by `[ ]`):

```
[ ] target11, target12, target13, ..., target1T [ ] target21, ...
[ ] input11,  input12,  input13,  ..., input1T  [ ] input21, ...
```

The mask tokens `[ ]` represent images with nothing but zeros.

For large datasets use Lua5.2 instead of LuaJIT to avoid memory errors (see torch.ch).

The following are attributes that can be set to `true` to modify the behavior of the loader:

- `cropeverystep` : samples a random uniform crop location every time-step (instead of once per sequence)
- `varyloadsize` : random-uniformly samples a `loadsize` between samplesize and loadsize (this effectively scales the cropped location)
- `scaleeverystep` : varies `loadsize` every step instead of once per sequence
- `randseq` : each new sequence is chosen random uniformly

loadMNIST

```
train, valid, test = dl.loadMNIST([datapath, validratio, scale,
srcurl])
```

Returns the training, validation and testing sets as 3 `TensorLoader` instances.
Each such loader encapsulates a part of the MNIST dataset which is
located in `datapath` (defaults to `dl.DATA_PATH/mnist`).
The `validratio` argument, a number between 0 and 1,
specifies the ratio of the 60000 training samples
that will be allocated to the validation set.
The `scale` argument specifies range within which pixel values will be scaled (defaults to `{0,1}`).
The `srcurl` specifies the URL from where the raw data can be downloaded from
if not located on disk.

loadCIFAR10

```
train, valid, test = dl.loadCIFAR10([datapath, validratio, scale,
srcurl])
```

Returns the training, validation and testing sets as 3 `TensorLoader` instances.
Each such loader encapsulates a part of the CIFAR10 dataset which is
located in `datapath` (defaults to `dl.DATA_PATH/cifar-10-batches-t7`).
The `validratio` argument, a number between 0 and 1,
specifies the ratio of the 50000 training samples

that will be allocated to the validation set.
The `scale` argument specifies range within which pixel values will be scaled (defaults to `{0,1}`).
The `srcurl` specifies the URL from where the raw data can be downloaded from
if not located on disk.

loadPTB

```
train, valid, test = dl.loadPTB(batchsize, [datapath, srcurl])
```

Returns the training, validation and testing sets as 3 `SequenceLoader` instance
Each such loader encapsulates a part of the Penn Tree Bank dataset which is
located in `datapath` (defaults to `dl.DATA_PATH/PennTreeBank`).
If the files aren't found in the `datapath`, they will be automatically downloaded
from the `srcurl` URL.
The `batchsize` specifies the number of samples that will be returned when
iterating through the dataset. If specified as a table, its elements
specify the `batchsize` of commensurate `train`, `valid` and `test` tables.
We recommend a `batchsize` of 1 for evaluation sets (e.g. `{50,1,1}`).

See RNNLM training script for an example.

loadImageNet

Ref.: A. http://image-net.org/challenges/LSVRC/2014/download-images-5jj5.php

```
train, valid = dl.loadImageNet(datapath, [nthread, loadsize,
samplesize, verbose])
```

Returns the training and validation sets of the Large Scale Visual Recognition Challenge 2014
(ILSVRC2014)
image classification dataset (commonly known as ImageNet).
The dataset hasn't changed from 2012-2014.

The returned `train` and `valid` loaders do not read all images into memory when first

loaded.
Each dataset is implemented using an ImageClass loader decorated by an AsyncIterator.

The `datapath` should point to a directory containing the outputs of the `downloadimagenet.lua` and `harmonizeimagenet.lua` scripts (see bellow).

Requirements

Due to its size, the data first needs to be prepared offline.
Use downloadimagenet.lua
to download and extract the data :

```
th downloadimagenet.lua --savePath '/path/to/diskspace/ImageNet'
```

The entire process requires about 360 GB of disk space to complete the download and extraction process.
This can be reduced to about 150 GB if the training set is downloaded and extracted first, and all the `.tar` files are manually deleted. Repeat for the validation set, devkit and metadata.
If you still don't have enough space in one partition, you can divide the data among different partitions.
We recommend a good internet connection (>60Mbs download) and a Solid-State Drives (SSD).

Use harmonizeimagenet.lua
to harmonize the train and validation sets:

```
th harmonizeimagenet.lua --dataPath /path/to/diskspace/ImageNet --
progress --forReal
```

Each set will then contain a directory of images for each class with name `class[id]` where `[id]` is a class index, between 1 and 1000, used for the ILVRC2014 competition.

Then we need to install graphicsmagick :

```
luarocks install graphicsmagick
```

Inference

As in the famous (Krizhevsky et al. 2012)
paper, the ImageNet training dataset samples images cropped from random
224x224 patches from the images resizes so that the smallest dimension has
size 256. As for the validation set, ten 224x224 patches are cropped per image,
i.e. center, four corners and their horizontal flips, and their predictions are averaged.

loadGBW

```
train, valid, test = dl.loadGBW(batchsize, [trainfile, datapath,
srcurl, verbose])
```

Loads the Google Billion Words corpus as MultiSequence loaders.
The preprocessing specified in
Google Billion Words language modeling benchmark
was applied to `training-monolingual.tokenized/news.20??.en.shuffled.tokenized` to generate the different subsets.
These subsets are automatically downloaded when not found on disk.
The task consists in predicting the next word given the previous ones.
The corpus contains approximately 30 million sentences of an average length of about 25 words.
In total, there are about 800 thousand (unique) words in the vocabulary, which makes it a very memory intensive problem.

loadSentiment140

```
train, valid, test = dl.loadSentiment140([datapath, minfreq,
seqlen, validratio, srcurl, progress])

Load & processing training data.
Number of tweets: 1600000
Vocabulary size: 155723
```

```
Number of occurences replaced with <OOV> token: 750575
Tweet corpus size (in number of tokens): 20061241
trainset set processed in 28.306740999222s
```

Load the Sentiment140 dataset.
This dataset can be used for sentiment analysis for microblogging websites like Twitter.
The task is to predict the sentiment of a tweet.
The input is a sequence of tokenized words with a default maximum sequence length of 50 (i.e. `seqlen=50`).
Targets can be one of three classes that map to the sentiment of the tweet: 1 = negative, 2 = neutral, 3 = positive. The neutral tweets are not present in the training data hence we ignore them from all (train, valid & test) datasets. This results in a 2-class (1=Negative, 2=Positive) dataset.
Tweets are tokenized using the `twitter/twokenize.py` script.
By default, only words with at least 3 occurrences (i.e. `minfreq=3`) in the training set are kept.
The dataset is automatically downloaded from `srcurl`, tokenized and parsed into a tensor the first time the loader is used.
The returned training, validation and test sets are encapsulated using the TensorLoader.
The input is padded with zeros before the tweet when it is shorted than `seqlen`.

The above is only printed when `progress=true` (the default) the first time the loader is invoked.
The processed data is subsequently cached to speedup future loadings.
To overwrite any cached data use `dl.overwrite=true`.

fitImageNormalize

```
ppf = dl.fitImageNormalize(trainset, [nsample, cachepath, verbose])
```

Returns a `ppf` preprocessing function that can be used to in-place normalize a batch of images (`inputs`)
channel-wise:

```
ppf(inputs)
```

The `trainset` argument is a DataLoader instance
containing image `inputs`. The mean and standard deviation will be measured
on `nsample` images (default 10000). When `cachepath` is provided, the

mean and standard deviation are saved for the next function call.