TNT Documentation

Release

Torch Contributors

May 02, 2018

Notes

TNT is a library providing powerful dataloading, logging and visualization utlities for Python. It is closely intergrated with PyTorch and is designed to enable rapid iteration with any model or training regimen.

CHAPTER 1

torchnet.dataset

Provides a `Dataset` interface, similar to vanilla PyTorch.

class torchnet.dataset.dataset.**Dataset**

Bases: object

batch(**args*, ***kwargs*)

parallel(**args*, ***kwargs*)

shuffle(**args*, ***kwargs*)

split(**args*, ***kwargs*)

transform(**args*, ***kwargs*)

1.1 BatchDataset

class torchnet.dataset.**BatchDataset**(*dataset*, *batchsize*, *perm=<function BatchDataset.<lambda>>*, *merge=None*, *policy='include-last'*, *filter=<function BatchDataset.<lambda>>*)

Bases: *torchnet.dataset.dataset.Dataset*

Dataset which batches the data from a given dataset.

Given a *dataset*, *BatchDataset* merges samples from this dataset to form a new sample which can be interpreted as a batch of size *batchsize*.

The *merge* function controls how the batching is performed. By default the occurrences are supposed to be tensors, and they aggregated along the first dimension.

It is often important to shuffle examples while performing the batch operation. *perm(idx, size)* is a function which returns the shuffled index of the sample at position *idx* in the underlying dataset. For convenience, the *size* of the underlying dataset is also passed to the function. By default, the function is the identity.

The underlying dataset size might or might not be always divisible by *batchsize*. The optional *policy* string specify how to handle corner cases.

Purpose: the concept of batch is problem dependent. In *torchnet*, it is up to the user to interpret a sample as a batch or not. When one wants to assemble samples from an existing dataset into a batch, then *BatchDataset* is suited for the job. Sometimes it is however more convenient to write a dataset from scratch providing "batched" samples.

Parameters

- **dataset** (*Dataset*) – Dataset to be batched.
- **batchsize** (*int*) – Size of the batch.
- **perm** (*function, optional*) – Function used to shuffle the dataset before batching. *perm(idx, size)* should return the shuffled index of *idx* th sample. By default, the function is the identity.
- **merge** (*function, optional*) – Function to control batching behaviour. *transform.makebatch(merge)* is used to make the batch. Default is None.
- **policy** (*str, optional*) – Policy to handle the corner cases when the underlying dataset size is not divisible by *batchsize*. One of (*include-last*, *skip-last*, *divisible-only*).
  - ***include-last* makes sure all samples of the underlying dataset** will be seen, batches will be of size equal or inferior to *batchsize*.
  - ***skip-last* will skip last examples of the underlying dataset if** its size is not properly divisible. Batches will be always of size equal to *batchsize*.
  - ***divisible-only* will raise an error if the underlying dataset** has not a size divisible by *batchsize*.
- **filter** (*function, optional*) – Function to filter the sample before batching. If *filter(sample)* is True, then sample is included for batching. Otherwise, it is excluded. By default, *filter(sample)* returns True for any *sample*.

1.2 ConcatDataset

class torchnet.dataset.**ConcatDataset**(*datasets*)

Bases: *torchnet.dataset.dataset.Dataset*

Dataset to concatenate multiple datasets.

Purpose: useful to assemble different existing datasets, possibly large-scale datasets as the concatenation operation is done in an on-the-fly manner.

Parameters **datasets** (*iterable*) – List of datasets to be concatenated

1.3 ListDataset

class torchnet.dataset.**ListDataset**(*elem_list*, *load=<function ListDataset.<lambda>>*, *path=None*)

Bases: *torchnet.dataset.dataset.Dataset*

Dataset which loads data from a list using given function.

Considering a *elem_list* (can be an iterable or a *string*) i-th sample of a dataset will be returned by *load(elem_list[i])*, where *load()* is a function provided by the user.

If *path* is provided, *elem_list* is assumed to be a list of strings, and each element *elem_list[i]* will prefixed by *path/* when fed to *load()*.

Purpose: many low or medium-scale datasets can be seen as a list of files (for example representing input samples). For this list of file, a target can be often inferred in a simple manner.

Parameters

- **elem_list** (*iterable/str*) – List of arguments which will be passed to *load* function. It can also be a path to file with each line containing the arguments to *load*
- **load** (*function, optional*) – Function which loads the data. i-th sample is returned by *load(elem_list[i])*. By default *load* is identity i.e, *lambda x: x*
- **path** (*str, optional*) – Defaults to None. If a string is provided, *elem_list* is assumed to be a list of strings, and each element *elem_list[i]* will prefixed by this string when fed to *load()*.

1.4 ResampleDataset

class torchnet.dataset.**ResampleDataset**(*dataset*, *sampler=<function ResampleDataset.<lambda>>*, *size=None*)

Bases: *torchnet.dataset.dataset.Dataset*

Dataset which resamples a given dataset.

Given a *dataset*, creates a new dataset which will (re-)sample from this underlying dataset using the provided *sampler(dataset, idx)* function.

If *size* is provided, then the newly created dataset will have the specified *size*, which might be different than the underlying dataset size. If *size* is not provided, then the new dataset will have the same size as the underlying one.

Purpose: shuffling data, re-weighting samples, getting a subset of the data. Note that an important sub-class *ShuffleDataset* is provided for convenience.

Parameters

- **dataset** (Dataset) – Dataset to be resampled.
- **sampler** (*function, optional*) – Function used for sampling. *idx'th sample is returned by 'dataset[sampler(dataset, idx)]*. By default *sampler(dataset, idx)* is the identity, simply returning *idx*. *sampler(dataset, idx)* must return an index in the range acceptable for the underlying *dataset*.
- **size** (*int, optional*) – Desired size of the dataset after resampling. By default, the new dataset will have the same size as the underlying one.

1.5 ShuffleDataset

class torchnet.dataset.**ShuffleDataset**(*dataset*, *size=None*, *replacement=False*)

Bases: torchnet.dataset.resampledataset.ResampleDataset

Dataset which shuffles a given dataset.

ShuffleDataset is a sub-class of *ResampleDataset* provided for convenience. It samples uniformly from the given *dataset* with, or without *replacement*. The chosen partition can be redrawn by calling *resample()*

If *replacement* is *true*, then the specified *size* may be larger than the underlying *dataset*. If *size* is not provided, then the new dataset size will be equal to the underlying *dataset* size.

Purpose: the easiest way to shuffle a dataset!

Parameters

- **dataset** (Dataset) – Dataset to be shuffled.
- **size** (*int, optional*) – Desired size of the shuffled dataset. If *replacement* is *true*, then can be larger than the *len(dataset)*. By default, the new dataset will have the same size as *dataset*.
- **replacement** (*bool, optional*) – True if uniform sampling is to be done with replacement. False otherwise. Defaults to false.

Raises ValueError – If *size* is larger than the size of the underlying dataset and *replacement* is False.

resample(*seed=None*)
Resample the dataset.

Parameters

- **seed** (*int, optional*) – Seed for resampling. By default no seed is
- **used.** –

1.6 SplitDataset

class torchnet.dataset.**SplitDataset**(*dataset*, *partitions*, *initial_partition=None*)
Bases: *torchnet.dataset.dataset.Dataset*

Dataset to partition a given dataset.

Partition a given *dataset*, according to the specified *partitions*. Use the method *select()* to select the current partition in use.

The *partitions* is a dictionary where a key is a user-chosen string naming the partition, and value is a number representing the weight (as a number between 0 and 1) or the size (in number of samples) of the corresponding partition.

Partioning is achieved linearly (no shuffling). See *ShuffleDataset* if you want to shuffle the dataset before partitioning.

Parameters

- **dataset** (Dataset) – Dataset to be split.
- **partitions** (*dict*) – Dictionary where key is a user-chosen string naming the partition, and value is a number representing the weight (as a number between 0 and 1) or the size (in number of samples) of the corresponding partition.
- **initial_partition** (*str, optional*) – Initial parition to be selected.

select(*partition*)
Select the parition.

Parameters **partition** (*str*) – Partition to be selected.

1.7 TensorDataset

class torchnet.dataset.**TensorDataset**(*data*)
Bases: *torchnet.dataset.dataset.Dataset*

Dataset from a tensor or array or list or dict.

TensorDataset provides a way to create a dataset out of the data that is already loaded into memory. It accepts data in the following forms:

tensor or numpy array *idx'th sample is 'data[idx]*

dict of tensors or numpy arrays *idx'th sample is '{k: v[idx] for k, v in data.items()}*

list of tensors or numpy arrays *idx'th sample is '[v[idx] for v in data]*

Purpose: Easy way to create a dataset out of standard data structures.

Parameters **data** (*dict/list/tensor/ndarray*) – Data for the dataset.

1.8 TransformDataset

class torchnet.dataset.**TransformDataset**(*dataset*, *transforms*)

Bases: *torchnet.dataset.dataset.Dataset*

Dataset which transforms a given dataset with a given function.

Given a function *transform*, and a *dataset*, *TransformDataset* applies the function in an on-the-fly manner when querying a sample with *__getitem__(idx)* and therefore returning *transform[dataset[idx]]*.

transform can also be a dict with functions as values. In this case, it is assumed that *dataset[idx]* is a dict which has all the keys in *transform*. Then, *transform[key]* is applied to dataset[idx][key] for each key in *transform*

The size of the new dataset is equal to the size of the underlying *dataset*.

Purpose: when performing pre-processing operations, it is convenient to be able to perform on-the-fly transformations to a dataset.

Parameters

- **dataset** (Dataset) – Dataset which has to be transformed.
- **transforms** (*function/dict*) – Function or dict with function as values. These functions will be applied to data.

CHAPTER 2

torchnet.engine

Engines are a utility to wrap a training loop. They provide several hooks which allow users to define their own fucntions to run at specified points during the train/val loop.

Some people like engines, others do not. TNT is build modularly, so you can use the other modules with/without using an engine.

2.1 torchnet.engine.Engine

class torchnet.engine.**Engine**

Bases: object

hook(*name*, *state*)

Registers a backward hook.

The hook will be called every time a gradient with respect to the Tensor is computed. The hook should have the following signature:

```
hook (grad) -> Tensor or None
```

The hook should not modify its argument, but it can optionally return a new gradient which will be used in place of `grad`. This function returns a handle with a method `handle.remove()` that removes the hook from the module.

Example

```
>>> v = torch.tensor([0., 0., 0.], requires_grad=True)
>>> h = v.register_hook(lambda grad: grad * 2)  # double the gradient
>>> v.backward(torch.tensor([1., 2., 3.]))
>>> v.grad
 2
 4
```

```
 6
[torch.FloatTensor of size (3,)]
>>> h.remove()  # removes the hook
```

test(*network*, *iterator*)

train(*network*, *iterator*, *maxepoch*, *optimizer*)

CHAPTER 3

torchnet.logger

Loggers provide a way to monitor your models. For example, the *MeterLogger* class provides easy meter visualizetion with Visdom , as well as the ability to print and save meters with the `ResultsWriter` class.

For visualization libraries, the current loggers support `Visdom`, although `TensorboardX` would also be simple to implement.

3.1 MeterLogger

class `torchnet.logger.`**MeterLogger**(*server='localhost'*, *env='main'*, *port=8097*, *title='DNN'*, *nclass=21*, *plotstylecombined=True*)

Bases: `object`

A class to package and visualize meters.

Parameters

- **server** – The uri of the Visdom server
- **env** – Visdom environment to log to.
- **port** – Port of the visdom server.
- **title** – The title of the MeterLogger. This will be used as a prefix for all plots.
- **nclass** – If logging for classification problems, the number of classes.
- **plotstylecombined** – Whether to plot train/test curves in the same window.

peek_meter()

Returns a dict of all meters and their values.

print_meter(*mode*, *iepoch*, *ibatch=1*, *totalbatch=1*, *meterlist=None*)

reset_meter(*iepoch*, *mode='Train'*)

update_loss(*loss*, *meter='loss'*)

update_meter(*output*, *target*, *meters={'accuracy'}*)

3.2 VisdomLogger

Logging to Visdom server

class torchnet.logger.visdomlogger.**BaseVisdomLogger**(*fields=None*, *win=None*, *env=None*, *opts={}*, *port=8097*, *server='localhost'*)

Bases: torchnet.logger.logger.Logger

The base class for logging output to Visdom.

THIS CLASS IS ABSTRACT AND MUST BE SUBCLASSED

Note that the Visdom server is designed to also handle a server architecture, and therefore the Visdom server must be running at all times. The server can be started with $ python -m visdom.server and you probably want to run it from screen or tmux.

log(**args*, ***kwargs*)

log_state(*state*)

Gathers the stats from self.trainer.stats and passes them into self.log, as a list

viz

class torchnet.logger.visdomlogger.**VisdomLogger**(*plot_type*, *fields=None*, *win=None*, *env=None*, *opts={}*, *port=8097*, *server='localhost'*)

Bases: *torchnet.logger.visdomlogger.BaseVisdomLogger*

A generic Visdom class that works with the majority of Visdom plot types.

log(**args*, ***kwargs*)

class torchnet.logger.visdomlogger.**VisdomPlotLogger**(*plot_type*, *fields=None*, *win=None*, *env=None*, *opts={}*, *port=8097*, *server='localhost'*, *name=None*)

Bases: *torchnet.logger.visdomlogger.BaseVisdomLogger*

log(**args*, ***kwargs*)

class torchnet.logger.visdomlogger.**VisdomSaver**(*envs=None*, *port=8097*, *server='localhost'*)

Bases: object

Serialize the state of the Visdom server to disk. Unless you have a fancy schedule, where different are saved with different frequencies, you probably only need one of these.

save(**args*, ***kwargs*)

class torchnet.logger.visdomlogger.**VisdomTextLogger**(*fields=None*, *win=None*, *env=None*, *opts={}*, *update_type='REPLACE'*, *port=8097*, *server='localhost'*)

Bases: *torchnet.logger.visdomlogger.BaseVisdomLogger*

Creates a text window in visdom and logs output to it.

The output can be formatted with fancy HTML, and it new output can be set to 'append' or 'replace' mode.

Parameters

- **fields** – Currently not used
- **update_type** – One of {'REPLACE', 'APPEND'}. Default 'REPLACE'.

For examples, make sure that your visdom server is running.

Example

```
>>> notes_logger = VisdomTextLogger(update_type='APPEND')
>>> for i in range(10):
>>>     notes_logger.log("Printing: {} of {}".format(i+1, 10))
# results will be in Visdom environment (default: http://localhost:8097)
```

log(*msg*, **args*, ***kwargs*)

valid_update_types = ['REPLACE', 'APPEND']

CHAPTER 4

torchnet.meter

Meters provide a way to keep track of important statistics in an online manner. TNT also provides convenient ways to visualize and manage meters via the *torchnet.logger.MeterLogger* class.

class torchnet.meter.meter.**Meter**

Meters provide a way to keep track of important statistics in an online manner.

This class is abstract, but provides a standard interface for all meters to follow.

add(*value*)

Log a new value to the meter

Parameters **value** – Next restult to include.

reset()

Resets the meter to default settings.

value()

Get the value of the meter in the current state.

4.1 Classification Meters

4.1.1 APMeter

class torchnet.meter.**APMeter**

The APMeter measures the average precision per class.

The APMeter is designed to operate on *NxK* Tensors *output* and *target*, and optionally a *Nx1* Tensor weight where (1) the *output* contains model output scores for *N* examples and *K* classes that ought to be higher when the model is more convinced that the example should be positively labeled, and smaller when the model believes the example should be negatively labeled (for instance, the output of a sigmoid function); (2) the *target* contains only values 0 (for negative examples) and 1 (for positive examples); and (3) the *weight* (> 0) represents weight for each sample.

add(*output*, *target*, *weight=None*)
Add a new observation

Parameters

- **output** (*Tensor*) – NxK tensor that for each of the N examples indicates the probability of the example belonging to each of the K classes, according to the model. The probabilities should sum to one over all classes
- **target** (*Tensor*) – binary NxK tensort that encodes which of the K classes are associated with the N-th input (eg: a row [0, 1, 0, 1] indicates that the example is associated with classes 2 and 4)
- **weight** (*optional, Tensor*) – Nx1 tensor representing the weight for each example (each weight > 0)

reset()
Resets the meter with empty member variables

value()
Returns the model's average precision for each class

Returns 1xK tensor, with avg precision for each class k

Return type ap (FloatTensor)

4.1.2 mAPMeter

class torchnet.meter.**mAPMeter**
The mAPMeter measures the mean average precision over all classes.

The mAPMeter is designed to operate on *NxK* Tensors *output* and *target*, and optionally a *Nx1* Tensor weight where (1) the *output* contains model output scores for *N* examples and *K* classes that ought to be higher when the model is more convinced that the example should be positively labeled, and smaller when the model believes the example should be negatively labeled (for instance, the output of a sigmoid function); (2) the *target* contains only values 0 (for negative examples) and 1 (for positive examples); and (3) the *weight* (> 0) represents weight for each sample.

4.1.3 ClassErrorMeter

class torchnet.meter.**ClassErrorMeter**(*topk=[1], accuracy=False*)

4.1.4 ConfusionMeter

class torchnet.meter.**ConfusionMeter**(*k*, *normalized=False*)
Maintains a confusion matrix for a given calssification problem.

The ConfusionMeter constructs a confusion matrix for a multi-class classification problems. It does not support multi-label, multi-class problems: for such problems, please use MultiLabelConfusionMeter.

Parameters

- **k** (*int*) – number of classes in the classification problem
- **normalized** (*boolean*) – Determines whether or not the confusion matrix is normalized or not

add(*predicted*, *target*)
Computes the confusion matrix of K x K size where K is no of classes

Parameters

- **predicted** (*tensor*) – Can be an N x K tensor of predicted scores obtained from the model for N examples and K classes or an N-tensor of integer values between 0 and K-1.
- **target** (*tensor*) – Can be a N-tensor of integer values assumed to be integer values between 0 and K-1 or N x K tensor, where targets are assumed to be provided as one-hot vectors

value()

Returns Confustion matrix of K rows and K columns, where rows corresponds to ground-truth targets and columns corresponds to predicted targets.

4.2 Regression/Loss Meters

4.2.1 AverageValueMeter

class torchnet.meter.**AverageValueMeter**

4.2.2 AUCMeter

class torchnet.meter.**AUCMeter**
The AUCMeter measures the area under the receiver-operating characteristic (ROC) curve for binary classification problems. The area under the curve (AUC) can be interpreted as the probability that, given a randomly selected positive example and a randomly selected negative example, the positive example is assigned a higher score by the classification model than the negative example.

The AUCMeter is designed to operate on one-dimensional Tensors *output* and *target*, where (1) the *output* contains model output scores that ought to be higher when the model is more convinced that the example should be positively labeled, and smaller when the model believes the example should be negatively labeled (for instance, the output of a signoid function); and (2) the *target* contains only values 0 (for negative examples) and 1 (for positive examples).

4.2.3 MovingAverageValueMeter

class torchnet.meter.**MovingAverageValueMeter**(*windowsize*)

4.2.4 MSEMeter

class torchnet.meter.**MSEMeter**(*root=False*)

4.3 Miscellaneous Meters

4.3.1 TimeMeter

class torchnet.meter.**TimeMeter**(*unit*)
<a name="TimeMeter"> #### tnt.TimeMeter(@ARGP) @ARGT

The *tnt.TimeMeter* is designed to measure the time between events and can be used to measure, for instance, the average processing time per batch of data. It is different from most other meters in terms of the methods it provides:

The *tnt.TimeMeter* provides the following methods:

- *reset()* resets the timer, setting the timer and unit counter to zero.
- *value()* returns the time passed since the last *reset()*; divided by the counter value when *unit=true*.

CHAPTER 5

torchnet.utils

5.1 MultiTaskDataLoader

class torchnet.utils.**MultiTaskDataLoader**(*datasets*, *batch_size=1*, *use_all=False*, ***loading_kwargs*)

Bases: object

Loads batches simultaneously from multiple datasets.

The MultiTaskDataLoader is designed to make multi-task learning simpler. It is ideal for jointly training a model for multiple tasks or multiple datasets. MultiTaskDataLoader is initialzes with an iterable of Dataset objects, and provides an iterator which will return one batch that contains an equal number of samples from each of the Dataset s.

Specifically, it returns batches of [(B_0, 0), (B_1, 1), ..., (B_k, k)] from datasets (D_0, ..., D_k), where each *B_i* has batch_size samples

Parameters

- **datasets** – A list of Dataset objects to serve batches from
- **batch_size** – Each batch from each Dataset will have this many samples
- **use_all** (*bool*) – If True, then the iterator will return batches until all datasets are exhausted. If False, then iteration stops as soon as one dataset runs out
- **loading_kwargs** – These are passed to the children dataloaders

Example

```
>>> train_loader = MultiTaskDataLoader([dataset1, dataset2], batch_size=3)
>>> for ((datas1, labels1), task1), (datas2, labels2), task2) in train_loader:
>>>     print(task1, task2)
0 1
0 1
```

```
...
0 1
```

5.2 ResultsWriter

class torchnet.utils.**ResultsWriter**(*filepath*, *overwrite=False*)

Bases: object

Logs results to a file.

The ResultsWriter provides a convenient interface for periodically writing results to a file. It is designed to capture all information for a given experiment, which may have a sequence of distinct tasks. Therefore, it writes results in the format:

```
{
    'tasks': [...]
    'results': [...]
}
```

The ResultsWriter class chooses to use a top-level list instead of a dictionary to preserve temporal order of tasks (by default).

Parameters

- **filepath** (*str*) – Path to write results to
- **overwrite** (*bool*) – whether to clobber a file if it exists

Example

```
>>> result_writer = ResultWriter(path)
>>> for task in ['CIFAR-10', 'SVHN']:
>>>    train_results = train_model()
>>>    test_results = test_model()
>>>    result_writer.update(task, {'Train': train_results, 'Test': test_results})
```

update(*task_name*, *result*)

Update the results file with new information.

Parameters

- **task_name** (*str*) – Name of the currently running task. A previously unseen `task_name` will create a new entry in both `tasks` and `results`.
- **result** – This will be appended to the list in `results` which corresponds to the `task_name` in `task_nametasks`.

5.3 Table module

torchnet.utils.table.**canmergetensor**(*tbl*)

torchnet.utils.table.**mergetensor**(*tbl*)

TNT was inspired by TorchNet, and legend says that it stood for "TorchNetTwo". Since then, TNT has developed on its own.

TNT provides simple methods to record model preformance in the torchnet.meter module and to log them to Visdom (or in the future, TensorboardX) with the torchnet.logging module.

In the future, TNT will also provide strong support for multi-task learning and transfer learning applications. It currently supports joint training data loading through torchnet.utils.MultiTaskDataLoader.

Python Module Index

t

Index

U

V