DN3

Release 0.0.1

Demetres Kostas

Nov 30, 2020

GUIDES

1 The Configuratron 3

2 Datasets 11

3 Metrics 13

4 Trainables 15

5 Transformations and Preprocessors 17

6 Configuratron 19

7 Datasets 23

8 Neural Network Building Blocks 31

9 Processes 39

10 Transforms 45

11 Indices and tables 47

Python Module Index 49

Index 51

This Python package is an effort to bridge the gap between the neuroscience library MNE-Python and the deep learning library PyTorch. This package's main focus is on minimizing boilerplate code, rapid deployment of known solutions, and increasing the reproducibility of *new deep-learning solutions* for the analysis of M/EEG data *(and may be compatible with other similar data… use at your own risk).*

Access to the code can be found at https://github.com/SPOClab-ca/dn3

The image above sketches out the structure of how the different modules of DN3 work together, but if you are new, we recommend starting with the *configuration guide*.

CHAPTER

ONE

THE CONFIGURATRON

High-level dataset and experiment descriptions

- *Why do I need this?*
- *A Little More Specific*
- *A Full Concrete Example*
- *That's great, but what's up with that 'architecture' entry?*
- *MOAR! I'm a power user*
- *Complete listing of configuratron (experiment level) options*
  - *Optional entries*
- *Complete listing of dataset configuration fields*
  - *Required entries*
  - *Special entries*
  - *Optional entries*
  - *Experimental/Risky Options*

1.1 Why do I need this?

Configuration files are perhaps where the advantages of DN3 are most apparent. Ostensibly, integrating *multiple* datasets to train a single process is as simple as loading files of each dataset from disk to be fed into a common deep learning training loop. The reality however, is rarely that simple. DN3 uses YAML formatted configuration files to streamline this process, and better organize the integration of *many* datasets.

Different file formats/extensions, sampling frequencies, directory structures make for annoying boilerplate with minor variations. Here (among other possible uses) a consistent configuration framework helps to automatically handle the variations across datatsets, for ease of integration down the road. If the dataset follows (or can be made to follow) the relatively generic directory structure of session instances nested in a single directory for each unique person, simply provided the top-level of this directory structure, a DN3 `Dataset` can be rapidly constructed, with easily adjustable *configuration* options.

Alternatively, if your dataset is all lumped into one folder, but follows a naming convention where the subject's name and the session id are embedded in a consistent naming format, e.g. *My-Data-S01-R0.edf* and *My-Data-S02-R1.edf*, two consistently formatted strings with two subjects (S01 and S02) and two runs (R0 and R1 - note that either subjects

or runs could also have been the same string and remained valid). In this case, you can use a (very *pythonic*) formatter to organize the data hierarchically: *filename_format: "My-Data-{subject}-{session}"*

1.2 A Little More Specific

Say we were evaluating a neural network architecture with some of our own data. We are happy with how it is currently working, but want to now evaluate it against a public dataset to compare with other work. Most of the time, this means writing a decent bit of code to load this new dataset. Instead, DN3 proposes that it should be as simple as:

```
public_dataset:
  toplevel: /path/to/the/files
```

As far as the real *configuration* aspect, perhaps this dataset has a unique time window for its trials? Is the dataset organized using filenames like the above "My-Data" example rather than directories? In that case:

```
public_dataset:
  toplevel: /path/to/the/files
  filename_format: "My-Data-{subject}-{session}"
  tmin: -0.1
  tlen: 1.5
```

Want to bandpass filter this data between 0.1Hz and 40Hz before use?

```
public_dataset:
  toplevel: /path/to/the/files
  filename_format: "My-Data-{subject}-{session}"
  tmin: -0.1
  tlen: 1.5
  hpf: 0.1
  lpf: 40
```

Hopefully this illustrates the advantage of organized datasets and configuration files, no boilerplate needed, you'll get nicely prepared and consistent dataset abstractions (see *Datasets*). Not only this, but it allows for people to share their *configurations*, for better reproducibility.

1.3 A Full Concrete Example

It takes a little more to make this a DN3 configuration, as we need to specify the existence of an experiment. Don't panic, it's as simple as adding an empty **Configuratron** to the yaml file that makes your configuration. Consider the contents of 'my_config.yml':

```
Configuratron:

datasets:
  in_house_dataset:
    name: "Awesome data"
    tmin: -0.5
    tlen: 1.5
    picks:
      - eeg
      - emg

  public_dataset:
```

(continues on next page)

(continued from previous page)

```
    toplevel: /path/to/the/files
    tmin: -0.1
    tlen: 1.5
    bandpass: [0.1, 40]

architecture:
  layers: 2
  activation: 'relu'
  dropout: 0.1
```

The important entry here is *Configuratron*, that confirms this is an entry-point for the configuratron, and *datasets* that lists the datasets we could use. The latter can either be named entries like the above, or a list of unnamed entries.

Now, on the python side of things:

```
from dn3.data.config import ExperimentConfig

experiment = ExperimentConfig("my_config.yml")
for ds_name, ds_config in experiment.datasets():
    dataset = ds_config.auto_construct_dataset()
    # Do some awesome things
```

The *dataset* variable above is now a DN3 *Dataset*, which now readily supports loading trials for training or separation according to people and/or sessions. Both the *in_house_dataset* and *public_dataset* will be available.

1.4 That's great, but what's up with that 'architecture' entry?

There isn't anything special to this, aside from providing a convenient location to add additional configuration values that one might need for a set of experiments. These fields will now be populated in the *experiment* variable above. So now, *experiment.architecture* is an object, with member variables populated from the yaml file.

1.5 MOAR! I'm a power user

One of the really cool (my Mom says so) aspects of the configuratron is the addition of !include directives. Aside from the top level of the file, you can include other files that can be readily reinterpreted as YAML, as supported by the pyyaml-include project. This means one could specify all the available datasets in one file called *datasets.yml* and include the complete listing for each configuration, say *config_shallow.yml* and *config_deep.yml* by saying *datasets: !include datasets.yml*. Or you could include JSON architecture configurations (potentially backed by your favourite cloud-based hyperparameter tracking module).

More directives might be added to the configuratron in the future, and we warmly welcome any suggestions/implementations others may come up with.

Further, that *Configuratron* entry above also allows for a variety of experiment-level options, which allows for common sets of channels, automatic adjustments of sampling frequencies and more. The trick is you need to keep reading.

1.6 Complete listing of configuratron (experiment level) options

1.6.1 Optional entries

use_only ***(list)*** A convenience option, whose purpose is to filter from datasets only the names in this list. This allows for inclusion of a large dataset file, and referencing certain named datasets. In this case, the names are the yaml key referencing the configuration.

deep1010 ***(bool)*** This will normalize and map all configuratron generated datasets using the `MappingDeep1010` transform. This is on by default.

samples ***(int)*** Providing samples will enforce a global (common) length across all datasets (probably want to use this in conjunction with the *sfreq* option below).

sfreq ***(float)*** Enforce a global sampling frequency, down or upsampling loaded sessions if necessary. If a session cannot be downsampled without aliasing (it violates the nyquist criterion), a warning message will be printed, and the session will be skipped.

preload ***(bool)*** Whether to preload recordings for all datasets. *This is overridden by individual *preload* options for dataset configurations.

trial_ids ***(bool)*** Whether to return an id (*long tensor*) for which trial *within each recording* each data sequence returned by the constructed dataset.

1.7 Complete listing of dataset configuration fields

1.7.1 Required entries

toplevel ***(required, directory)*** Specifies the toplevel directory of the dataset.

1.7.2 Special entries

filename_format ***(str)*** The special entry will assume that after scanning for all the correct *type* of file, the *subject* and *session* (or in DN3-speak, the *Thinker* and *Recording*) name can be parsed from the filepath. This should be a python-*format*-style string with two required substrings: *{subject}* and *{session}* that form a template for parsing subject and session ids from the path. Note, the file extension should not be included, and fixed length can be specified by trailing *:N* for length *N*, e.g. *{subject:2}* for specifically 2 characters devoted to subject ID.

The next few entries are superseded by the *Configuratron* entry *samples*, which defines a global number of samples parameter. If this is not the case, **one of the following two is required**.

tlen ***(required, float)*** The length of time to use for each retrieved datapoint. If *epoched* trials (see `EpochTorchRecording`) are required, *tmin* must also be specified.

samples ***(required-ish, float)*** As an alternative to tlen, for when you want to align datasets with pretty similar sampling frequencies, you can specify samples. If used, tlen is ignored (and not needed) and is inferred from the number of samples desired.

1.7.3 Optional entries

tmin ***(float)*** If specified, epochs the recordings into trials at each event (can be modified by *events* config below) onset with respect to *tmin*. So if *tmin* is negative, happens before the event marker, positive is after, and 0 is at the onset.

baseline ***(list, None)*** This option will only be used with epoched data (tmin is specified). This is simply propagated to the Epoch's constructor as is. Where *None* can be specified using a tilde character: ~, as in *baseline: [~, ~]* to use all data for basline subtraction. **Unlike the default constructor, here by default, no baseline correction is performed.**

events ***(list, map/dict)*** This can be formatted in one of three ways:

1. Unspecified - all events parsed by find_events(), falling-back to events_from_annotations()
2. A list of event numbers that filter the set found from the above.
3. A list of events (keys) and then labels (values) for those events, which filters as above, e.g.:

```
events:
  T1: 5
  T2: 6
```

The values should be integer codes, if both sides are numeric, this is used to map stim channel events to new values, otherwise (if the keys are strings), the annotations are searched.

In all cases, the codes from the stim channel or annotations will not in fact correspond to the subsequent labels loaded. This is because the labels don't necessarily fit a minimal spanning set starting with 0. In other words, if I had say, 4 labels, they are not guaranteed to be 0, 1, 2 and 3 as is needed for loss functions downstream.

The latter two configuration options above *do however* provide some control over this, with the order of the listed events corresponding to the index of the used label. e.g. *left_hand* and *right_hand* above have class labels 0 and 1 respectively.

If the reasoning for the above is not clear, not to worry. Just know you can't assume that annotated event 1 is label 1. Instead use `EpochTorchRecording.get_mapping()` to resolve labels to the original annotations or event codes.

annotation_format ***(str)*** In some cases, annotations may be provided as *separate* (commonly edf) files. This string should specify how to match the annotation file, optionally using the subject and session ids. This uses standard unix-style pattern matching, augmented with the ability to specify the subject with *{subject(:...)}* and the session with *{session(:...)}* as is used by filename_format. So one could use a pattern like: *"Data--*{subject}*-annotation"*. ***Note, now by default, any file matching the annotation pattern is also excluded from being loaded as raw data.**

targets ***(int)*** The number of targets to classify if there are events. This is inferred otherwise.

chunk_duration ***(float)*** If specified, rather than using event offsets, create events every chunk_duration seconds, and then still use **tlen** and **tmin** with respect to these events. *This works with annotated recordings, and not recordings that rely on `stim` channels.*

picks ***(list)*** This option can take two forms:

- The names of the desired channels
- Channel types as used by MNE's pick_types()

By default, will select only eeg and meg channels (if meg, will try to automatically resolve as described here)

exclude_channels ***(list)*** This is similar to the above, except it is a list of *nix pattern match exclusions. Which means it can be the channel names (that you want to exclude) themselves, or use wildards such as "FT*" or, "F[!39]".

The first excludes all channels beginning with FT, the second, excludes all channels beginning with F *except* F3 and F9.

rename_channels ***(dict)*** Using this option, key's are the **new** name, and values are *nix-style pattern matching strings for the old channel names. *Warning* if an old channel matches to multiple new ones, new channel used is selected arbitrarily. Renaming is performed **before** exclusion.

decimate ***(bool)*** Only works with epoch data, must be > 0, default 1. Amount to decimate trials.

name ***(string)*** A more human-readable name for the dataset. This should be used to describe the dataset itself, not one of (potentially) many different configurations of said dataset (which might all share this parameter).

preload ***(bool)*** Whether to preload the recordings from this dataset. This overrides the experiment level *preload* option. Note that not all data formats support *preload*: False, but most do.

hpf ***(float)*** This entry (and the very similar *lpf* option) provide an option to highpass filter the raw data before anything else. It also supercedes any `preload`ing options, as the data needs to be loaded to perform this. It is specified in Hz.

lpf ***(float)*** This entry (and the very similar *hpf* option) provide an option to lowpass filter the raw data before anything else. It also supercedes any `preload`ing options, as the data needs to be loaded to perform this. It is specified in Hz.

extensions ***(list)*** The file extensions to seek out when searching for sessions in the dataset. These should include the '.', as in '.edf' . *This can include extensions not handled by auto_construction. A handler must then be provided using* `DatasetConfig.add_extension_handler()`

stride ***(int)*** Only for `RawTorchRecording`. The number of samples to slide forward for the next section of raw data. Defaults to 1, which means that each sample in the recording (aside from the last `sample_length - 1`) is used as the beginning of a retrieved section.

drop_bad ***(bool)*** Whether to ignore any events annotated as bad. Defaults to *False*

data_max ***(float, bool)*** The maximum value taken by any recording in the dataset. Providing a float will assume this value, setting this to *True* instead automatically determines this value when loading data. These are required for a fully-specified use of the Deep1010 mapping.

CAUTION: this can be extremely slow. If specified, the value will be printed and should probably be explicitly added to the configuration subsequently.

data_min ***(float, bool)*** The minimum value taken by any recording in the dataset. Providing a float will assume this value, setting this to *True* instead automatically determines this value when loading data. These are required for a fully-specified use of the Deep1010 mapping.

CAUTION: this can be extremely slow. If specified, the value will be printed and should probably be explicitly added to the configuration subsequently.

dataset_id ***(int)*** This allows datasets to be given specific ids. By default, none are provided. If set to an int, this dataset will have this integer `dataset_id.

exclude_people ***(list)*** List of people (identified by the name of their respective directories) to be ignored. Supports Unix-style pattern matching *within quotations* (*, ?, [seq], [!seq]).

exclude_sessions ***(list)*** List of sessions (files) to be ignored when performing automatic constructions. Supports Unix-style pattern matching *within quotations* (*, ?, [seq], [!seq]).

exclude ***(map/dict)*** This is a more extensively formatted version of *exclude_people* and *exclude_sessions* from above. Here, people, sessions and timespans (specified in seconds) can be excluded using a hierarchical representation. The easiest way to understand this is by example. Consider:

```
exclude_people:
  - Person01
```

(continues on next page)

(continued from previous page)

```
exclude:
  Person02: ~
  Person03:
    Session01: ~
  Person04:
    Session01:
      - [0, 0.5]
    Session02:
      - [0, 0.5]
      - [100, 120]
```

The above says that *Person01* and *Person02* should both be completely ignored. *Session01* from *Person03* should be similarly ignored (with any other *Person03* session left available). Finally for *Person04* the data between 0 and 0.5 seconds of *Session01* in addition to both the times between 0 and 0.5 and 100 and 120 seconds from *Session02* should be ignored. If

In summary, it allows more fine-grained exclusion **without pattern matching**, and can be used in conjunction with the other exclusion options. For those familiar with MNE's *bads* system, it is not used here, this allows for config files to be shared rather than annotated copies of the original data. Further, this allows for easier by-hand editing.

1.7.4 Experimental/Risky Options

load_onthefly ***(bool)*** This overrides any preload values (for the dataset or experiment) and minimizes memory overhead from recordings at the cost of compute time and increased disk I/O. This is only really helpful if you have a dataset *so large* that mne's Raw instances start to fill your memory (this is not impossible, so if you are running out of memory, try switching on this option). Currently this does not work with epoched data.

CHAPTER

TWO

DATASETS

- *Returning IDs*

For the most part, DN3 datasets are a simple wrapping around MNE's Epoch and Raw objects, with the intent of:

1. Providing a common API to minimize boilerplate around common divisions of data at the session, person and dataset boundaries
2. Encouraging more consistency in data loading across multiple projects.
3. Integration of (CPU bound) transformations operations, executed *on-the-fly* during deep network training

Thus there are three main interfaces:

1. The `Recording` classes
   - *`RawTorchRecording`* and *`EpochTorchRecording`*
2. The *`Thinker`* class, that collects a set of a *single person's* sessions
3. The *`Dataset`* class, that collects a set of multiple *`Thinker`* that performed the same task under the same relevant context *(which in most cases means all the subjects of an experiment)*.

2.1 Returning IDs

At many levels of abstraction, particularly for *`Thinker`* and *`Dataset`* , there is the option of returning identifying values for the context of the trial within the larger dataset. In other words, the session, person, dataset, and task ids can also be acquired while iterating through these datasets. These will always be returned sandwiched between the actual recording value (first) and (if epoched) the class label for the recording (last), from *most general* to *most specific*. Consider this example iteration over a *`Dataset`*:

```
dataset = Dataset(thinkers, dataset_id=10, task_id=15, return_session_id=True, return_
↪person_id=True,
                  return_dataset_id=True, return_task_id=True)

for i, (x, task_id, ds_id, person_id, session_id, y) in enumerate(dataset):
    awesome_stuff()
```

CHAPTER

THREE

METRICS

Metrics are how your work gets evaluated. Here we talk about some of the tools DN3 provides to do this well.

- *sklearn*

3.1 sklearn

CHAPTER

FOUR

TRAINABLES

Trainables cover two branches, `Processes` and `Models`. They both could consist of potentially learnable parameters, but largely, `Processes` are the way in which *Models* are trained. Thus, `Processes` consist of a training loop, optimizer, loss function(s), *metrics*,

CHAPTER

FIVE

TRANSFORMATIONS AND PREPROCESSORS

- *Summary*
- *Instance Transforms*
- *Batch Transforms*
- *Multiple Worker Processes Warning*
- *Preprocessors*

5.1 Summary

One of the advantages of using PyTorch as the underlying computation library, is eager graph execution that can leverage native python. In other words, it lets us integrate arbitrary operations in a largely parallel fashion to our training (*particularly if we are using the GPU for any neural networks*).

5.2 Instance Transforms

Enter the `InstanceTransform` and its subclasses. When added to a *`Dataset`*, these perform operations on each fetched recording sequence, be it a trial or cropped sequence of raw data. For the most part, they are simply callable objects, implementing `__call__()` to modify a `Tensor` unless they modify the number/representation of channels, sampling frequency or sequence length of the data.

They are specifically *instance* transforms, because they do not transform more than a single crop of data (from a single person and dataset). This means, that these are done before a batch is aggregated for training. If the transform results in many differently shaped tensors, **a batch will not properly be created, so watch out for that**!

5.3 Batch Transforms

These are the exceptions that prove the `InstanceTransform` rule. These transforms operate only *after* data has been aggregated into a batch, and it is just about to be fed into a network for training (or otherwise). These are attached to trainable *`Processess`* instead of *`Datasets`*.

5.4 Multiple Worker Processes Warning

After attaching enough transforms, you may find that, even with most of the deep learning side being done on the GPU loading the training data may become the bottleneck.

5.5 Preprocessors

Preprocessor (s) on the other hand are a method to *create* a transform after first encountering all of the Recordings of a *Dataset*. Simply put, if the transform is known *a priori*, the BaseTransform interface is sufficient. Otherwise, a *Preprocessor* can be used to both modify Recordings in place *before* training, and create a transformation to modify sequences *on-the-fly*.

CHAPTER

SIX

CONFIGURATRON

See the *configuration guide* for a detailed listing of configuration options.

Classes

DatasetConfig(name, config[, ...])	Parses dataset entries in DN3 config
ExperimentConfig(config_filename[, ...])	Parses DN3 configuration files.
RawOnTheFlyRecording(*args, **kwds)	

class dn3.configuratron.config.**DatasetConfig**(*name: str*, *config: dict*, *adopt_auxiliaries=True*, *ext_handlers=None*, *deep1010=None*, *samples=None*, *sfreq=None*, *preload=False*, *return_trial_ids=False*)

Parses dataset entries in DN3 config **Methods**

add_custom_raw_loader(custom_loader)	This is used to provide a custom implementation of taking a filename, and returning a `mne.io.Raw()` instance.
add_custom_thinker_loader(thinker_loader)	Add custom code to load a specific thinker from a set of session files.
add_extension_handler(extension, handler)	Provide callable code to create a raw instance from sessions with certain file extensions.
add_progress_callbacks([session_callback, ...])	Add callbacks to be invoked on successful loading of session and/or thinker.
auto_construct_dataset([mapping])	This creates a dataset using the config values.
auto_mapping([files, reset_exclusions])	Generates a mapping of sessions and people of the dataset, assuming files are stored in the structure:
scan_toplevel()	Scan the provided toplevel for all files that may belong to the dataset.

add_custom_raw_loader(*custom_loader*)

This is used to provide a custom implementation of taking a filename, and returning a `mne.io.Raw()` instance. If properly constructed, all further configuratron options, such as resampling, epoching, filtering etc. should occur automatically.

This is used to load unconventional files, e.g. '.mat' files from matlab, or custom '.npy' arrays, etc.

Notes

Consider using `mne.io.Raw.add_events()` to integrate otherwise difficult (for the configuratron) to better specify events for each recording.

Parameters **custom_loader** (*callable*) – A function that expects a single `pathlib.Path()` instance as argument and returns an instance of `mne.io.Raw()`. To gracefully ignore problematic sessions, raise `DN3ConfigException` within.

add_custom_thinker_loader(*thinker_loader*)
Add custom code to load a specific thinker from a set of session files.

Warning: For all intents and purposes, this circumvents most of the configuratron, and results in it being mostly a tool for organizing dataset files. Most of the options are not leveraged and must be implemented by the custom loader. Please open an issue if you'd like to develop this option further!

Parameters **thinker_loader** – A function that takes a list argument that consists of the filenames (str) of all the detected session for the given thinker and a second argument for the detected name of the person. The function should return a single instance of type *Thinker*. To gracefully ignore the person, raise a `DN3ConfigException`

add_extension_handler(*extension: str*, *handler*)
Provide callable code to create a raw instance from sessions with certain file extensions. This is useful for handling of custom file formats, while preserving a consistent experiment framework.

Parameters

- **extension** (*str*) – An extension that includes the '.', e.g. '.csv'
- **handler** (*callable*) – Callback with signature f(path_to_file: str) -> mne.io.Raw

add_progress_callbacks(*session_callback=None*, *thinker_callback=None*)
Add callbacks to be invoked on successful loading of session and/or thinker. Optionally, these can modify the respective loaded instances.

Parameters

- **session_callback** – A function that expects a single session argument and can modify the (or return an alternative) session.
- **thinker_callback** – The same as for session, but with Thinker instances.

auto_construct_dataset(*mapping=None*, ***dsargs*)
This creates a dataset using the config values. If tlen and tmin are specified in the config, creates epoched dataset, otherwise Raw.

Parameters

- **mapping** (*dict, optional*) – A dict specifying a list of sessions (as paths to files) for each person_id in the dataset. e.g. {

  person_1: [sess_1.edf, ...], person_2: [sess_1.edf], ...

  } If not specified, will use *auto_mapping()* to generate.
- **dsargs** – Any additional arguments to feed for the creation of the dataset. i.e. keyword arguments to *Dataset*'s constructor (which id's to return). If *dataset_info* is provided here, it will override what was inferrable from the configuration file.

Returns **dataset** – An instance of *Dataset*, constructed according to mapping.

Return type *Dataset*

auto_mapping (*files=None*, *reset_exclusions=True*)
Generates a mapping of sessions and people of the dataset, assuming files are stored in the structure: *toplevel*/(*optional - <version>)/<person-id>/<session-id>.{ext}

Parameters **files** (*list*) – Optional list of files (convertible to *Path* objects, e.g. relative or absolute strings) to be used. If not provided, will use *scan_toplevel()*.

Returns **mapping** – The keys are of all the people in the dataset, and each value another similar mapping to that person's sessions.

Return type dict

scan_toplevel ()
Scan the provided toplevel for all files that may belong to the dataset.

Returns **files** – A listing of all the candidate filepaths (before excluding those that match exclusion criteria).

Return type list

class dn3.configuratron.config.**ExperimentConfig** (*config_filename: str*, *adopt_auxiliaries=True*)
Parses DN3 configuration files. Checking the DN3 token for listed datasets.

class dn3.configuratron.config.**RawOnTheFlyRecording** (**args*, ***kwds*)
Methods

preprocess(preprocessor[, apply_transform])	Applies a preprocessor to the dataset

preprocess (*preprocessor*, *apply_transform=True*)
Applies a preprocessor to the dataset

Parameters

- **preprocessor** (Preprocessor) – A preprocessor to be applied
- **apply_transform** (*bool*) – Whether to apply the transform to this dataset (and all members e.g thinkers or sessions) after preprocessing them. Alternatively, the preprocessor is returned for manual application of its transform through `Preprocessor.get_transform()`

Returns **preprocessor** – The preprocessor after application to all relevant thinkers

Return type *Preprocessor*

CHAPTER

SEVEN

DATASETS

Classes

DN3ataset(*args, **kwds)	
Dataset(*args, **kwds)	Collects thinkers, each of which may collect multiple recording sessions of the same tasks, into a dataset with
DatasetInfo(dataset_name[, data_max, ...])	This objects contains non-critical meta-data that might need to be tracked for **:py:`Dataset`** objects.
EpochTorchRecording(*args, **kwds)	
RawTorchRecording(*args, **kwds)	Interface for bridging mne Raw instances as PyTorch compatible "Dataset".
Thinker(*args, **kwds)	Collects multiple recordings of the same person, intended to be of the same task, at different times or conditions.

class dn3.data.dataset.**DN3ataset**(**args*, ***kwds*)

Methods

add_transform(transform)	Add a transformation that is applied to every fetched item in the dataset
clear_transforms()	Remove all added transforms from dataset.
clone()	A copy of this object to allow the repetition of recordings, thinkers, etc.
preprocess(preprocessor[, apply_transform])	Applies a preprocessor to the dataset
to_numpy([batch_size, num_workers])	Commits the dataset to numpy-formatted arrays.

Attributes

channels	returns: **channels** – The channel sets used by the dataset.
sequence_length	returns: **sequence_length** – The length of each instance in number of samples
sfreq	returns: **sampling_frequency** – The sampling frequencies employed by the dataset.

add_transform(*transform*)

Add a transformation that is applied to every fetched item in the dataset

Parameters **transform** (*BaseTransform*) – For each item retrieved by __getitem__, transform is called to modify that item.

property channels
returns: **channels** – The channel sets used by the dataset. :rtype: list

clear_transforms()
Remove all added transforms from dataset.

clone()
A copy of this object to allow the repetition of recordings, thinkers, etc. that load data from the same memory/files but have their own tracking of ids.

Returns **cloned** – New copy of this object.

Return type *DN3ataset*

preprocess(*preprocessor:* dn3.transforms.preprocessors.Preprocessor, *apply_transform=True*)
Applies a preprocessor to the dataset

Parameters

- **preprocessor** (Preprocessor) – A preprocessor to be applied
- **apply_transform** (*bool*) – Whether to apply the transform to this dataset (and all members e.g thinkers or sessions) after preprocessing them. Alternatively, the preprocessor is returned for manual application of its transform through `Preprocessor.get_transform()`

Returns **preprocessor** – The preprocessor after application to all relevant thinkers

Return type *Preprocessor*

property sequence_length
returns: **sequence_length** – The length of each instance in number of samples :rtype: int, list

property sfreq
returns: **sampling_frequency** – The sampling frequencies employed by the dataset. :rtype: float, list

to_numpy(*batch_size=64*, *batch_transforms: list = None*, *num_workers=4*, ***dataloader_kwargs*)
Commits the dataset to numpy-formatted arrays. Useful for saving dataset to disk, or preparing for tools that expect numpy-formatted data rather than iteratable.

Notes

A pytorch `DataLoader` is used to fetch the data to conveniently leverage multiprocessing, and naturally

Parameters

- **batch_size** (*int*) – The number of items to fetch per worker. This probably doesn't need much tuning.
- **num_workers** (*int*) – The number of spawned processes to fetch and transform data.
- **batch_transforms** (*list*) – These are potential batch-level transforms that
- **dataloader_kwargs** (*dict*) – Keyword arguments for the pytorch `DataLoader` that underpins the fetched data

Returns **data** – A list of numpy arrays.

Return type list

class dn3.data.dataset.**Dataset**(**args*, ***kwds*)
Collects thinkers, each of which may collect multiple recording sessions of the same tasks, into a dataset with (largely) consistent:

Methods

add_transform(transform[, thinkers])	Add a transformation that is applied to every fetched item in the dataset
clear_transforms()	Remove all added transforms from dataset.
dump_dataset(toplevel[, apply_transforms])	Dumps the dataset to the file location specified by toplevel, with a single file per session made of all the return tensors (as numpy data) loaded by the dataset.
get_sessions()	Accumulates all the sessions from each thinker in the dataset in a nested dictionary.
get_targets()	Collect all the targets (i.e.
get_thinkers()	Accumulates a consistently ordered list of all the thinkers in the dataset.
lmso([folds, test_splits, validation_splits])	This *generates* a "Leave-multiple-subject-out" (LMSO) split.
loso([validation_person_id, test_person_id])	This *generates* a "Leave-one-subject-out" (LOSO) split.
preprocess(preprocessor[, apply_transform, ...])	Applies a preprocessor to the dataset
safe_mode([mode])	This allows switching *safe_mode* on or off.
update_id_returns([trial, session, person, ...])	Updates which ids are to be returned by the dataset.

Attributes

channels	returns: **channels** – The channel sets used by the dataset.
sequence_length	returns: **sequence_length** – The length of each instance in number of samples
sfreq	returns: **sampling_frequency** – The sampling frequencies employed by the dataset.

- hardware: - channel number/labels - sampling frequency
- annotation paradigm: - consistent event types

add_transform(*transform*, *thinkers=None*)
Add a transformation that is applied to every fetched item in the dataset

Parameters **transform** (*BaseTransform*) – For each item retrieved by __getitem__, transform is called to modify that item.

property channels
returns: **channels** – The channel sets used by the dataset. :rtype: list

clear_transforms()
Remove all added transforms from dataset.

dump_dataset(*toplevel*, *apply_transforms=True*)
Dumps the dataset to the file location specified by toplevel, with a single file per session made of all the return tensors (as numpy data) loaded by the dataset.

Parameters

- **toplevel** (*str*) – The toplevel location to dump the dataset to. This folder (and path) will be created if it does not exist. Each person will have a subdirectory therein, with

numpy-formatted files for each session within that.

- **apply_transforms** (*bool*) – Whether to apply the transforms while preparing the data to be saved.

get_sessions()
Accumulates all the sessions from each thinker in the dataset in a nested dictionary.

Returns **session_dict** – Keys are the thinkers of *get_thinkers()*, values are each another dictionary that maps session ids to _Recording

Return type dict

get_targets()
Collect all the targets (i.e. labels) that this Thinker's data is annotated with.

Returns **targets** – A numpy-formatted array of all the targets/label for this thinker.

Return type np.ndarray

get_thinkers()
Accumulates a consistently ordered list of all the thinkers in the dataset. It is this order that any automatic segmenting through *loso()* and *lmso()* will be done.

Returns **thinker_names**

Return type list

lmso(*folds=10*, *test_splits=None*, *validation_splits=None*)
This *generates* a "Leave-multiple-subject-out" (LMSO) split. In other words X-fold cross-validation, with boundaries enforced at thinkers (each person's data is not split into different folds).

Parameters

- **folds** (*int*) – If this is specified and *splits* is None, will split the subjects into this many folds, and then use each fold as a test set in turn (and the previous fold - starting with the last - as validation).
- **test_splits** (*list, tuple*) –

  This should be a list of tuples/lists of either:

  - The ids of the consistent test set. In which case, folds must be specified, or validation_splits is a nested list that .
  - Two sub lists, first testing, second validation ids

Yields

- **training** (*Dataset*) – Another dataset that represents the training set
- **validation** (*Dataset*) – The validation people as a dataset
- **test** (*Thinker*) – The test people as a dataset

loso(*validation_person_id=None*, *test_person_id=None*)
This *generates* a "Leave-one-subject-out" (LOSO) split. Tests each person one-by-one, and validates on the previous (the first is validated with the last).

Parameters

- **validation_person_id** (*(int, str, list, optional)*) – If specified, and corresponds to one of the person_ids in this dataset, the loso cross validation will consistently generate this thinker as *validation*. If *list*, must be the same length as *test_person_id*, say a length N. If so, will yield N each in sequence, and use remainder for test.

- **test_person_id** (*(int, str, list, optional)*) – Same as *validation_person_id*, but for testing. However, testing may be a list when validation is a single value. Thus if testing is N ids, will yield N values, with a consistent single validation person. If a single id (int or str), and *validation_person_id* is not also a single id, will ignore *validation_person_id* and loop through all others that are not the *test_person_id*.

Yields

- **training** (*Dataset*) – Another dataset that represents the training set
- **validation** (*Thinker*) – The validation thinker
- **test** (*Thinker*) – The test thinker

preprocess(*preprocessor: dn3.transforms.preprocessors.Preprocessor*, *apply_transform=True*, *thinkers=None*)

Applies a preprocessor to the dataset

Parameters

- **preprocessor** (Preprocessor) – A preprocessor to be applied
- **thinkers** (*(None, Iterable)*) – If specified (default is None), the thinkers to use for preprocessing calculation
- **apply_transform** (*bool*) – Whether to apply the transform to this dataset (all thinkers, not just those specified for preprocessing) after preprocessing them. Exclusive application to specific thinkers can be done using the return value and a separate call to *add_transform* with the same *thinkers* list.

Returns **preprocessor** – The preprocessor after application to all relevant thinkers

Return type *Preprocessor*

safe_mode(*mode=True*)

This allows switching *safe_mode* on or off. When safe_mode is on, if data is ever NaN, it is captured before being returned and a report is generated.

Parameters **mode** (*bool*) – The status of whether in safe mode or not.

property sequence_length

returns: **sequence_length** – The length of each instance in number of samples :rtype: int, list

property sfreq

returns: **sampling_frequency** – The sampling frequencies employed by the dataset. :rtype: float, list

update_id_returns(*trial=None*, *session=None*, *person=None*, *task=None*, *dataset=None*)

Updates which ids are to be returned by the dataset. If any argument is *None* it preserves the previous value.

Parameters

- **trial** (*None, bool*) – Whether to return trial ids.
- **session** (*None, bool*) – Whether to return session ids.
- **person** (*None, bool*) – Whether to return person ids.
- **task** (*None, bool*) – Whether to return task ids.
- **dataset** (*None, bool*) – Whether to return dataset ids.

class dn3.data.dataset.**DatasetInfo**(*dataset_name*, *data_max=None*, *data_min=None*, *excluded_people=None*, *targets=None*)

This objects contains non-critical meta-data that might need to be tracked for **:py:`Dataset`** objects. Generally not necessary to be constructed manually, these are created by the configuratron to automatically create transforms and/or other processes downstream.

class dn3.data.dataset.**EpochTorchRecording**(**args*, ***kwds*)

Methods

event_mapping()	Maps the labels returned by this to the events as recorded in the original annotations or stim channel.
preprocess(preprocessor[, apply_transform])	Applies a preprocessor to the dataset

event_mapping()

Maps the labels returned by this to the events as recorded in the original annotations or stim channel.

Returns **mapping** – Keys are the class labels used by this object, values are the original event signifier.

Return type dict

preprocess(*preprocessor:* dn3.transforms.preprocessors.Preprocessor, *apply_transform=True*)

Applies a preprocessor to the dataset

Parameters

- **preprocessor** (*Preprocessor*) – A preprocessor to be applied
- **apply_transform** (*bool*) – Whether to apply the transform to this dataset (and all members e.g thinkers or sessions) after preprocessing them. Alternatively, the preprocessor is returned for manual application of its transform through `Preprocessor.get_transform()`

Returns **preprocessor** – The preprocessor after application to all relevant thinkers

Return type *Preprocessor*

class dn3.data.dataset.**RawTorchRecording**(**args*, ***kwds*)

Interface for bridging mne Raw instances as PyTorch compatible "Dataset".

Parameters

- **raw** (*mne.io.Raw*) – Raw data, data does not need to be preloaded.
- **tlen** (*float*) – Length of recording specified in seconds.
- **session_id** (*(int, str, optional)*) – A unique (with respect to a thinker within an eventual dataset) identifier for the current recording session. If not specified, defaults to '0'.
- **person_id** (*(int, str, optional)*) – A unique (with respect to an eventual dataset) identifier for the particular person being recorded.
- **stride** (*int*) – The number of samples to skip between each starting offset of loaded samples.

Methods

preprocess(preprocessor[, apply_transform])	Applies a preprocessor to the dataset

preprocess(*preprocessor:* dn3.transforms.preprocessors.Preprocessor, *apply_transform=True*)

Applies a preprocessor to the dataset

Parameters

- **preprocessor** (*Preprocessor*) – A preprocessor to be applied
- **apply_transform** (*bool*) – Whether to apply the transform to this dataset (and all members e.g thinkers or sessions) after preprocessing them. Alternatively, the preprocessor is returned for manual application of its transform through `Preprocessor.get_transform()`

Returns **preprocessor** – The preprocessor after application to all relevant thinkers

Return type *Preprocessor*

class dn3.data.dataset.**Thinker**(**args*, ***kwds*)

Collects multiple recordings of the same person, intended to be of the same task, at different times or conditions.

Methods

add_transform(transform)	Add a transformation that is applied to every fetched item in the dataset
clear_transforms([deep_clear])	Remove all added transforms from dataset.
get_targets()	Collect all the targets (i.e.
preprocess(preprocessor[, apply_transform, ...])	Applies a preprocessor to the dataset
split([training_sess_ids, ...])	Split the thinker's data into training, validation and testing sets.

Attributes

channels	returns: **channels** – The channel sets used by the dataset.
sequence_length	returns: **sequence_length** – The length of each instance in number of samples
sfreq	returns: **sampling_frequency** – The sampling frequencies employed by the dataset.

add_transform(*transform*)

Add a transformation that is applied to every fetched item in the dataset

Parameters **transform** (*BaseTransform*) – For each item retrieved by __getitem__, transform is called to modify that item.

property channels

returns: **channels** – The channel sets used by the dataset. :rtype: list

clear_transforms(*deep_clear=False*)

Remove all added transforms from dataset.

get_targets()

Collect all the targets (i.e. labels) that this Thinker's data is annotated with.

Returns **targets** – A numpy-formatted array of all the targets/label for this thinker.

Return type np.ndarray

preprocess(*preprocessor: dn3.transforms.preprocessors.Preprocessor*, *apply_transform=True*, *sessions=None*)

Applies a preprocessor to the dataset

Parameters

- **preprocessor** (*Preprocessor*) – A preprocessor to be applied
- **sessions** (*(None, Iterable)*) – If specified (default is None), the sessions to use for preprocessing calculation
- **apply_transform** (*bool*) – Whether to apply the transform to this dataset (all sessions, not just those specified for preprocessing) after preprocessing them. Exclusive application to select sessions can be done using the return value and a separate call to *add_transform* with the same *sessions* list.

Returns **preprocessor** – The preprocessor after application to all relevant thinkers

Return type *Preprocessor*

property sequence_length

returns: **sequence_length** – The length of each instance in number of samples :rtype: int, list

property sfreq

returns: **sampling_frequency** – The sampling frequencies employed by the dataset. :rtype: float, list

split(*training_sess_ids=None*, *validation_sess_ids=None*, *testing_sess_ids=None*, *test_frac=0.25*, *validation_frac=0.25*)

Split the thinker's data into training, validation and testing sets.

Parameters

- **test_frac** (*float*) – Proportion of the total data to use for testing, this is overridden by *testing_sess_ids*.
- **validation_frac** (*float*) – Proportion of the data remaining - after removing test proportion/sessions - to use as validation data. Likewise, *validation_sess_ids* overrides this value.
- **training_sess_ids** (*: (Iterable, None)*) – The session ids to be explicitly used for training.
- **validation_sess_ids** (*(Iterable, None)*) – The session ids to be explicitly used for validation.
- **testing_sess_ids** (*(Iterable, None)*) – The session ids to be explicitly used for testing.

Returns

- **training** (*DN3ataset*) – The training dataset
- **validation** (*DN3ataset*) – The validation dataset
- **testing** (*DN3ataset*) – The testing dataset

CHAPTER

EIGHT

NEURAL NETWORK BUILDING BLOCKS

DN3 provides a variety of ready-made networks and building blocks (any PyTorch modules would suffice) to be trained.

8.1 Models

Classes

Classifier(targets, samples, channels[, ...])	A generic Classifer container.
DN3BaseModel(samples, channels[, ...])	This is a base model used by the provided models in the library that is meant to make those included in this library as powerful and multi-purpose as is reasonable.
EEGNet(targets, samples, channels[, do, ...])	This is the DN3 re-implementation of Lawhern et.
EEGNetStrided(targets, samples, channels[, ...])	This is the DN3 re-implementation of Lawhern et.
LogRegNetwork(targets, samples, channels[, ...])	In effect, simply an implementation of linear kernel (multi)logistic regression
StrideClassifier(targets, samples, channels)	
TIDNet(targets, samples, channels[, ...])	The Thinker Invariant Densenet from Kostas & Rudzicz 2020, https://doi.org/10.1088/1741-2552/abb7a7

class dn3.trainable.models.**Classifier**(*targets*, *samples*, *channels*, *return_features=True*)

A generic Classifer container. This container breaks operations up into feature extraction and feature classification to enable convenience in transfer learning and more. **Methods**

forward(*x)	Defines the computation performed at every call.
freeze_features([unfreeze, freeze_classifier])	In many cases, the features learned by a model in one domain can be applied to another case.
from_dataset(dataset, **modelargs)	Create a classifier from a dataset.
make_new_classification_layer()	This allows for a distinction between the classification layer(s) and the rest of the network.

forward(**x*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks

while the latter silently ignores them.

---

freeze_features(*unfreeze=False*, *freeze_classifier=False*)

In many cases, the features learned by a model in one domain can be applied to another case.

This method freezes (or un-freezes) all but the *classifier* layer. So that any further training does not (or does if unfreeze=True) affect these weights.

Parameters

- **unfreeze** (*bool*) – To unfreeze weights after a previous call to this.
- **freeze_classifier** (*bool*) – Commonly, the classifier layer will not be frozen (default). Setting this to *True* will freeze this layer too.

classmethod from_dataset(*dataset:* dn3.data.dataset.DN3ataset, ***modelargs*)

Create a classifier from a dataset.

Parameters

- **dataset** –
- **modelargs** (*dict*) – Options to construct the dataset, if dataset does not have listed targets, targets must be specified in the keyword arguments or will fall back to 2.

Returns **model** – A new *Classifier* ready to classifiy data from *dataset*

Return type *Classifier*

make_new_classification_layer()

This allows for a distinction between the classification layer(s) and the rest of the network. Using a basic formulation of a network being composed of two parts feature_extractor & classifier.

This method is for implementing the classification side, so that methods like *freeze_features()* works as intended.

Anything besides a layer that just flattens anything incoming to a vector and Linearly weights this to the target should override this method, and there should be a variable called *self.classifier*

class dn3.trainable.models.**DN3BaseModel**(*samples*, *channels*, *return_features=True*)

This is a base model used by the provided models in the library that is meant to make those included in this library as powerful and multi-purpose as is reasonable.

It is not strictly necessary to have new modules inherit from this, any nn.Module should suffice, but it provides some integrated conveniences...

The premise of this model is that deep learning models can be understood as *learned pipelines*. These *DN3BaseModel* objects, are re-interpreted as a two-stage pipeline, the two stages being *feature extraction* and *classification*. **Methods**

clone()	This provides a standard way to copy models, weights and all.
forward(x)	Defines the computation performed at every call.

clone()

This provides a standard way to copy models, weights and all.

forward(*x*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

class dn3.trainable.models.**EEGNet**(*targets*, *samples*, *channels*, *do=0.25*, *pooling=8*, *F1=8*, *D=2*, *t_len=65*, *F2=16*, *return_features=False*)

This is the DN3 re-implementation of Lawhern et. al.'s EEGNet from: https://iopscience.iop.org/article/10.1088/1741-2552/aace8c

Notes

The implementation below is in no way officially sanctioned by the original authors, and in fact is missing the constraints the original authors have on the convolution kernels, and may or may not be missing more...

That being said, in *our own personal experience*, this implementation has fared no worse when compared to implementations that include this constraint (albeit, those were *also not written* by the original authors).

class dn3.trainable.models.**EEGNetStrided**(*targets*, *samples*, *channels*, *do=0.25*, *pooling=8*, *F1=8*, *D=2*, *t_len=65*, *F2=16*, *return_features=False*, *stride_width=2*)

This is the DN3 re-implementation of Lawhern et. al.'s EEGNet from: https://iopscience.iop.org/article/10.1088/1741-2552/aace8c

Notes

The implementation below is in no way officially sanctioned by the original authors, and in fact is missing the constraints the original authors have on the convolution kernels, and may or may not be missing more...

That being said, in *our own personal experience*, this implementation has fared no worse when compared to implementations that include this constraint (albeit, those were *also not written* by the original authors).

class dn3.trainable.models.**LogRegNetwork**(*targets*, *samples*, *channels*, *return_features=True*)

In effect, simply an implementation of linear kernel (multi)logistic regression

class dn3.trainable.models.**StrideClassifier**(*targets*, *samples*, *channels*, *stride_width=2*, *return_features=False*)

Methods

make_new_classification_layer()	This allows for a distinction between the classification layer(s) and the rest of the network.

make_new_classification_layer()

This allows for a distinction between the classification layer(s) and the rest of the network. Using a basic formulation of a network being composed of two parts feature_extractor & classifier.

This method is for implementing the classification side, so that methods like `freeze_features()` works as intended.

Anything besides a layer that just flattens anything incoming to a vector and Linearly weights this to the target should override this method, and there should be a variable called *self.classifier*

class dn3.trainable.models.**TIDNet**(*targets*, *samples*, *channels*, *s_growth=24*, *t_filters=32*, *do=0.4*, *pooling=20*, *activation=<class 'torch.nn.modules.activation.LeakyReLU'>*, *temp_layers=2*, *spat_layers=2*, *temp_span=0.05*, *bottleneck=3*, *summary=-1*, *return_features=False*)

The Thinker Invariant Densenet from Kostas & Rudzicz 2020, https://doi.org/10.1088/1741-2552/abb7a7

This alone is not strictly "thinker invariant", but on average outperforms shallower models at inter-subject prediction capability.

8.2 Layers

Classes

Concatenate([axis])	
ConvBlock2D(in_filters, out_filters, kernel)	Implements complete convolution block with order:
DenseFilter(in_features, growth_rate[, ...])	
DenseSpatialFilter(channels, growth, depth)	
Expand([axis])	
Flatten()	
IndexSelect(indices)	
Permute(axes)	
SpatialFilter(channels, filters, depth[, ...])	
Squeeze([axis])	
TemporalFilter(channels, filters, depth, ...)	

class dn3.trainable.layers.**Concatenate**(*axis=-1*)

Methods

forward(*x)	Defines the computation performed at every call.

forward(**x*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

class dn3.trainable.layers.**ConvBlock2D**(*in_filters*, *out_filters*, *kernel*, *stride=(1, 1)*, *padding=0*, *dilation=1*, *groups=1*, *do_rate=0.5*, *batch_norm=True*, *activation=<class 'torch.nn.modules.activation.LeakyReLU'>*, *residual=False*)

Implements complete convolution block with order:

- Convolution
- dropout (spatial)
- activation

- batch-norm
- (optional) residual reconnection

Methods

forward(input, **kwargs)	Defines the computation performed at every call.

forward(*input*, ***kwargs*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

class dn3.trainable.layers.**DenseFilter**(*in_features*, *growth_rate*, *filter_len=5*, *do=0.5*, *bottleneck=2*, *activation=<class 'torch.nn.modules.activation.LeakyReLU'>*, *dim=-2*)

Methods

forward(x)	Defines the computation performed at every call.

forward(*x*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

class dn3.trainable.layers.**DenseSpatialFilter**(*channels*, *growth*, *depth*, *in_ch=1*, *bottleneck=4*, *dropout_rate=0.0*, *activation=<class 'torch.nn.modules.activation.LeakyReLU'>*, *collapse=True*)

Methods

forward(x)	Defines the computation performed at every call.

forward(*x*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

class dn3.trainable.layers.**Expand**(*axis=- 1*)

Methods

forward(x)	Defines the computation performed at every call.

forward(*x*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

class dn3.trainable.layers.**Flatten**

Methods

forward(x)	Defines the computation performed at every call.

forward(*x*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

class dn3.trainable.layers.**IndexSelect**(*indices*)

Methods

forward(*x)	Defines the computation performed at every call.

forward(**x*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

class dn3.trainable.layers.**Permute**(*axes*)

Methods

forward(x)	Defines the computation performed at every call.

forward(*x*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

class dn3.trainable.layers.**SpatialFilter**(*channels*, *filters*, *depth*, *in_ch=1*, *dropout_rate=0.0*, *activation=<class 'torch.nn.modules.activation.LeakyReLU'>*, *batch_norm=True*, *residual=False*)

Methods

forward(x)	Defines the computation performed at every call.

forward(*x*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

class dn3.trainable.layers.**Squeeze**(*axis=- 1*)

Methods

forward(x)	Defines the computation performed at every call.

forward(*x*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

class dn3.trainable.layers.**TemporalFilter**(*channels*, *filters*, *depth*, *temp_len*, *dropout=0.0*, *activation=<class 'torch.nn.modules.activation.LeakyReLU'>*, *residual='netwise'*)

Methods

forward(x)	Defines the computation performed at every call.

forward(*x*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks

while the latter silently ignores them.

CHAPTER

NINE

PROCESSES

Processes describe how a (or several) neural networks (trainables) are trained with given data. In a large number of cases, simply leveraging `StandardClassification` will be sufficient for many cases, as it provides many idiosynchratic options, as listed below.

See the processes guide for an overview on how these work.

Classes

`BaseProcess`([lr, metrics, ...])	Initialization of the Base Trainable object.
`LDAMLoss`(cls_num_list[, max_m, weight, s])	
`StandardClassification`(classifier[, ...])	

Functions

`get_label_balance`(dataset)	Given a dataset, return the proportion of each target class and the counts of each class type

class dn3.trainable.processes.**BaseProcess**(*lr=0.001*, *metrics=None*, *evaluation_only_metrics=None*, *l2_weight_decay=0.01*, *cuda=None*, ***kwargs*)

Initialization of the Base Trainable object. Any learning procedure that leverages DN3atasets should subclass this base class.

By default uses the SGD with momentum optimization. **Methods**

`build_network`(**kwargs)	This method is used to add trainable modules to the process.
`calculate_loss`(inputs, outputs)	Given the inputs to and outputs from underlying modules, calculate the loss.
`calculate_metrics`(inputs, outputs)	Given the inputs to and outputs from the underlying module.
`evaluate`(dataset, **loader_kwargs)	Calculate and return metrics for a dataset
`fit`(training_dataset[, epochs, ...])	sklearn/keras-like convenience method to simply proceed with training across multiple epochs of the provided
`forward`(*inputs)	Given a batch of inputs, return the outputs produced by the trainable module.
`load_best`(best)	Load the parameters as saved by `save_best()`.
`parameters`()	All the trainable parameters in the Trainable.

continues on next page

Table 3 – continued from previous page

predict(dataset, **loader_kwargs)	Determine the outputs for all loaded data from the dataset
save_best()	Create a snapshot of what is being currently trained for re-laoding with the `load_best()` method.
set_scheduler(scheduler[, step_every_batch])	This allow the addition of a learning rate schedule to the process.

build_network(***kwargs*)
This method is used to add trainable modules to the process. Rather than placing objects for training in the __init__ method, they should be placed here.

By default any arguments that propagate unused from __init__ are included here.

calculate_loss(*inputs*, *outputs*)
Given the inputs to and outputs from underlying modules, calculate the loss.

Returns Single loss quantity to be minimized.

Return type Loss

calculate_metrics(*inputs*, *outputs*)
Given the inputs to and outputs from the underlying module. Return tracked metrics.

Parameters

- **inputs** – Input tensors.
- **outputs** – Output tensors.

Returns **metrics** – Dictionary of metric quantities.

Return type OrderedDict, None

evaluate(*dataset*, ***loader_kwargs*)
Calculate and return metrics for a dataset

Parameters

- **dataset** (*DN3ataset, DataLoader*) – The dataset that will be used for evaluation, if not a DataLoader, one will be constructed
- **loader_kwargs** (*dict*) – Args that will be passed to the dataloader, but *shuffle* and *drop_last* will be both be forced to *False*

Returns **metrics** – Metric scores for the entire

Return type OrderedDict

fit(*training_dataset*, *epochs=1*, *validation_dataset=None*, *step_callback=None*, *resume_epoch=None*, *resume_iteration=None*, *log_callback=None*, *epoch_callback=None*, *batch_size=8*, *warmup_frac=0.2*, *retain_best='loss'*, *validation_interval=None*, *train_log_interval=None*, ***loader_kwargs*)
sklearn/keras-like convenience method to simply proceed with training across multiple epochs of the provided dataset

Parameters

- **training_dataset** (*DN3ataset, DataLoader*) –
- **validation_dataset** (*DN3ataset, DataLoader*) –
- **epochs** (*int*) – Total number of epochs to fit

- **resume_epoch** (*int*) – The starting epoch to train from. This will likely only be used to resume training at a certain point.
- **resume_iteration** (*int*) – Similar to start epoch but specified in batches. This can either be used alone, or in conjunction with *start_epoch*. If used alone, the start epoch is the floor of *start_iteration* divided by batches per epoch. In other words this specifies cumulative batches if start_epoch is not specified, and relative to the current epoch otherwise.
- **step_callback** (*callable*) – Function to run after every training step that has signature: fn(train_metrics) -> None
- **log_callback** (*callable*) – Function to run after every log interval that has signature: fn(train_metrics) -> None
- **epoch_callback** (*callable*) – Function to run after every epoch that has signature: fn(validation_metrics) -> None
- **batch_size** (*int*) – The batch_size to be used for the training and validation datasets. This is ignored if they are provided as *DataLoader*.
- **warmup_frac** (*float*) – The fraction of iterations that will be spent *increasing* the learning rate under the default 1cycle policy (with cosine annealing). Value will be automatically clamped values between [0, 0.5]
- **retain_best** (*(str, None)*) – **If `validation_dataset` is provided**, which model weights to retain. If 'loss' (default), will retain the model at the epoch with the lowest validation loss. If another string, will assume that is the metric to monitor for the *highest score*. If None, the final model is used.
- **validation_interval** (*int, None*) – The number of batches between checking the validation dataset
- **train_log_interval** (*int, None*) – The number of batches between persistent logging of training metrics, if None (default) happens at the end of every epoch.
- **loader_kwargs** – Any remaining keyword arguments will be passed as such to any DataLoaders that are automatically constructed. If both training and validation datasets are provided as *DataLoaders*, this will be ignored.

Notes

If the datasets above are provided as DN3atasets, automatic optimizations are performed to speed up loading. These include setting the number of workers = to the number of CPUs/system threads - 1, and pinning memory for rapid CUDA transfer if leveraging the GPU. Unless you are very comfortable with PyTorch, it's probably better to not provide your own DataLoader, and let this be done automatically.

Returns

- **train_log** (*Dataframe*) – Metrics after each iteration of training as a pandas dataframe
- **validation_log** (*Dataframe*) – Validation metrics after each epoch of training as a pandas dataframe

forward(**inputs*)

Given a batch of inputs, return the outputs produced by the trainable module.

Parameters **inputs** – Tensors needed for underlying module.

Returns Outputs of module

Return type outputs

load_best(*best*)
Load the parameters as saved by *save_best()*.

Parameters **best** (*Any*) –

parameters()
All the trainable parameters in the Trainable. This includes any architecture parameters and meta-parameters.

Returns An iterator of parameters

Return type params

predict(*dataset*, ***loader_kwargs*)
Determine the outputs for all loaded data from the dataset

Parameters

- **dataset** (*DN3ataset, DataLoader*) – The dataset that will be used for evaluation, if not a DataLoader, one will be constructed
- **loader_kwargs** (*dict*) – Args that will be passed to the dataloader, but *shuffle* and *drop_last* will be both be forced to *False*

Returns

- **inputs** (*Tensor*) – The exact inputs used to calculate the outputs (in case they were stochastic and need saving)
- **outputs** (*Tensor*) – The outputs from each run of **:function:`forward`**

save_best()
Create a snapshot of what is being currently trained for re-laoding with the *load_best()* method.

Returns **best** – Whatever format is needed for *load_best()*, will be the argument provided to it.

Return type Any

set_scheduler(*scheduler*, *step_every_batch=False*)
This allow the addition of a learning rate schedule to the process. By default, a linear warmup with cosine decay will be used. Any scheduler that is an instance of `Scheduler` (pytorch's schedulers, or extensions thereof) can be set here. Additionally, a string keywords can be used including:

- "constant"

Parameters

- **scheduler** (*str, Scheduler*) –
- **step_every_batch** (*bool*) – Whether to call step after every batch (if *True*), or after every epoch (*False*)

class dn3.trainable.processes.**LDAMLoss**(*cls_num_list*, *max_m=0.5*, *weight=None*, *s=30*)
Methods

forward(x, target)	Defines the computation performed at every call.

forward(*x*, *target*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

class dn3.trainable.processes.**StandardClassification**(*classifier: torch.nn.modules.module.Module*, *loss_fn=None*, *cuda=None*, *metrics=None*, *learning_rate=0.01*, *label_smoothing=None*, ***kwargs*)

Methods

calculate_loss(inputs, outputs)	Given the inputs to and outputs from underlying modules, calculate the loss.
fit(training_dataset[, epochs, ...])	sklearn/keras-like convenience method to simply proceed with training across multiple epochs of the provided
forward(*inputs)	Given a batch of inputs, return the outputs produced by the trainable module.

calculate_loss(*inputs*, *outputs*)
Given the inputs to and outputs from underlying modules, calculate the loss.

Returns Single loss quantity to be minimized.

Return type Loss

fit(*training_dataset*, *epochs=1*, *validation_dataset=None*, *step_callback=None*, *epoch_callback=None*, *batch_size=8*, *warmup_frac=0.2*, *retain_best='loss'*, *balance_method=None*, ***loader_kwargs*)
sklearn/keras-like convenience method to simply proceed with training across multiple epochs of the provided dataset

Parameters

- **training_dataset** (*DN3ataset, DataLoader*) –
- **validation_dataset** (*DN3ataset, DataLoader*) –
- **epochs** (*int*) –
- **step_callback** (*callable*) – Function to run after every training step that has signature: fn(train_metrics) -> None
- **epoch_callback** (*callable*) – Function to run after every epoch that has signature: fn(validation_metrics) -> None
- **batch_size** (*int*) – The batch_size to be used for the training and validation datasets. This is ignored if they are provided as *DataLoader*.
- **warmup_frac** (*float*) – The fraction of iterations that will be spent *increasing* the learning rate under the default 1cycle policy (with cosine annealing). Value will be automatically clamped values between [0, 0.5]
- **retain_best** (*(str, None)*) – **If `validation_dataset` is provided**, which model weights to retain. If 'loss' (default), will retain the model at the epoch with

the lowest validation loss. If another string, will assume that is the metric to monitor for the *highest score*. If None, the final model is used.

- **balance_method** (*(None, str)*) – If and how to balance training samples when training. *None* (default) will simply randomly sample all training samples equally. 'undersample' will sample each class N_min times where N_min is equal to the number of examples in the minority class. 'oversample' will sample each class N_max times, where N_max is the number of the majority class.
- **loader_kwargs** – Any remaining keyword arguments will be passed as such to any DataLoaders that are automatically constructed. If both training and validation datasets are provided as *DataLoaders*, this will be ignored.

Notes

If the datasets above are provided as DN3atasets, automatic optimizations are performed to speed up loading. These include setting the number of workers = to the number of CPUs/system threads - 1, and pinning memory for rapid CUDA transfer if leveraging the GPU. Unless you are very comfortable with PyTorch, it's probably better to not provide your own DataLoader, and let this be done automatically.

Returns

- **train_log** (*Dataframe*) – Metrics after each iteration of training as a pandas dataframe
- **validation_log** (*Dataframe*) – Validation metrics after each epoch of training as a pandas dataframe

forward(**inputs*)

Given a batch of inputs, return the outputs produced by the trainable module.

Parameters **inputs** – Tensors needed for underlying module.

Returns Outputs of module

Return type outputs

dn3.trainable.processes.**get_label_balance**(*dataset*)

Given a dataset, return the proportion of each target class and the counts of each class type

Parameters **dataset** –

Returns

Return type sample_weights, counts

CHAPTER

TEN

TRANSFORMS

- *Instance Transforms*
- *Batch Transforms*
- *Preprocessors*

10.1 Instance Transforms

Classes

FixedScale([low_bound, high_bound])	Scale the input to range from low to high
MappingDeep1010(dataset[, add_scale_ind, ...])	Maps various channel sets into the Deep10-10 scheme, and normalizes data between [-1, 1] with an additional scaling parameter to describe the relative scale of a trial with respect to the entire dataset.
NoisyBlankDeep1010([mask_index, purge_mask])	
ZScore([only_trial_data])	Z-score normalization of trials

Functions

same_channel_sets(channel_sets)	Validate that all the channel sets are consistent, return false if not

class dn3.transforms.instance.**FixedScale**(*low_bound=- 1*, *high_bound=1*)

Scale the input to range from low to high

class dn3.transforms.instance.**MappingDeep1010**(*dataset*, *add_scale_ind=True*, *return_mask=False*)

Maps various channel sets into the Deep10-10 scheme, and normalizes data between [-1, 1] with an additional scaling parameter to describe the relative scale of a trial with respect to the entire dataset.

TODO - refer to eventual literature on this **Methods**

new_channels(old_channels)	This is an optional method that indicates the transformation modifies the representation and/or presence of channels.

new_channels (*old_channels: numpy.ndarray*)
This is an optional method that indicates the transformation modifies the representation and/or presence of channels.

Parameters **old_channels** (*ndarray*) – An array whose last two dimensions are channel names and channel types.

Returns **new_channels** – An array with the channel names and types after this transformation. Supports the addition of dimensions e.g. a list of channels into a rectangular grid, but the *final two dimensions* must remain the channel names, and types respectively.

Return type ndarray

class dn3.transforms.instance.**NoisyBlankDeep1010** (*mask_index=1*, *purge_mask=False*)

class dn3.transforms.instance.**ZScore** (*only_trial_data=True*)
Z-score normalization of trials

dn3.transforms.instance.**same_channel_sets** (*channel_sets: list*)
Validate that all the channel sets are consistent, return false if not

10.2 Batch Transforms

10.3 Preprocessors

Classes

Preprocessor()	Base class for various preprocessing actions.

class dn3.transforms.preprocessors.**Preprocessor**
Base class for various preprocessing actions. Sub-classes are called with a subclass of *_Recording* and operate on these instances in-place.

Any modifications to data specifically should be implemented through a subclass of `BaseTransform`, and returned by the method *get_transform()* **Methods**

get_transform()	Generate and return any transform associated with this preprocessor.

get_transform()
Generate and return any transform associated with this preprocessor. Should be used after applying this to a dataset, i.e. through `DN3ataset.preprocess()`

Returns **transform**

Return type BaseTransform

CHAPTER

ELEVEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

d

dn3.configuratron.config, 19
dn3.data.dataset, 23
dn3.trainable.layers, 34
dn3.trainable.models, 31
dn3.trainable.processes, 39
dn3.transforms.batch, 46
dn3.transforms.instance, 45
dn3.transforms.preprocessors, 46

INDEX

A

add_custom_raw_loader() (*dn3.configuratron.config.DatasetConfig method*), 19
add_custom_thinker_loader() (*dn3.configuratron.config.DatasetConfig method*), 20
add_extension_handler() (*dn3.configuratron.config.DatasetConfig method*), 20
add_progress_callbacks() (*dn3.configuratron.config.DatasetConfig method*), 20
add_transform() (*dn3.data.dataset.Dataset method*), 25
add_transform() (*dn3.data.dataset.DN3ataset method*), 23
add_transform() (*dn3.data.dataset.Thinker method*), 29
auto_construct_dataset() (*dn3.configuratron.config.DatasetConfig method*), 20
auto_mapping() (*dn3.configuratron.config.DatasetConfig method*), 21

B

BaseProcess (*class in dn3.trainable.processes*), 39
build_network() (*dn3.trainable.processes.BaseProcess method*), 39

C

calculate_loss() (*dn3.trainable.processes.BaseProcess method*), 40
calculate_loss() (*dn3.trainable.processes.StandardClassification method*), 43
calculate_metrics() (*dn3.trainable.processes.BaseProcess method*), 40
channels() (*dn3.data.dataset.Dataset property*), 25
channels() (*dn3.data.dataset.DN3ataset property*), 23
channels() (*dn3.data.dataset.Thinker property*), 29
Classifier (*class in dn3.trainable.models*), 31
clear_transforms() (*dn3.data.dataset.Dataset method*), 25
clear_transforms() (*dn3.data.dataset.DN3ataset method*), 24
clear_transforms() (*dn3.data.dataset.Thinker method*), 29
clone() (*dn3.data.dataset.DN3ataset method*), 24
clone() (*dn3.trainable.models.DN3BaseModel method*), 32
Concatenate (*class in dn3.trainable.layers*), 34
ConvBlock2D (*class in dn3.trainable.layers*), 34

D

Dataset (*class in dn3.data.dataset*), 24
DatasetConfig (*class in dn3.configuratron.config*), 19
DatasetInfo (*class in dn3.data.dataset*), 27
DenseFilter (*class in dn3.trainable.layers*), 35
DenseSpatialFilter (*class in dn3.trainable.layers*), 35
dn3.configuratron.config module, 19
dn3.data.dataset module, 23
dn3.trainable.layers module, 34
dn3.trainable.models module, 31
dn3.trainable.processes module, 39
dn3.transforms.batch module, 46
dn3.transforms.instance module, 45
dn3.transforms.preprocessors module, 46
DN3ataset (*class in dn3.data.dataset*), 23
DN3BaseModel (*class in dn3.trainable.models*), 32
dump_dataset() (*dn3.data.dataset.Dataset method*), 25

E

EEGNet (*class in dn3.trainable.models*), 33
EEGNetStrided (*class in dn3.trainable.models*), 33
EpochTorchRecording (*class in dn3.data.dataset*), 28
evaluate() (*dn3.trainable.processes.BaseProcess method*), 40
event_mapping() (*dn3.data.dataset.EpochTorchRecording method*), 28
Expand (*class in dn3.trainable.layers*), 35
ExperimentConfig (*class in dn3.configuratron.config*), 21

F

fit() (*dn3.trainable.processes.BaseProcess method*), 40
fit() (*dn3.trainable.processes.StandardClassification method*), 43
FixedScale (*class in dn3.transforms.instance*), 45
Flatten (*class in dn3.trainable.layers*), 36
forward() (*dn3.trainable.layers.Concatenate method*), 34
forward() (*dn3.trainable.layers.ConvBlock2D method*), 35
forward() (*dn3.trainable.layers.DenseFilter method*), 35
forward() (*dn3.trainable.layers.DenseSpatialFilter method*), 35
forward() (*dn3.trainable.layers.Expand method*), 36
forward() (*dn3.trainable.layers.Flatten method*), 36
forward() (*dn3.trainable.layers.IndexSelect method*), 36
forward() (*dn3.trainable.layers.Permute method*), 36
forward() (*dn3.trainable.layers.SpatialFilter method*), 37
forward() (*dn3.trainable.layers.Squeeze method*), 37
forward() (*dn3.trainable.layers.TemporalFilter method*), 37
forward() (*dn3.trainable.models.Classifier method*), 31
forward() (*dn3.trainable.models.DN3BaseModel method*), 32
forward() (*dn3.trainable.processes.BaseProcess method*), 41
forward() (*dn3.trainable.processes.LDAMLoss method*), 42
forward() (*dn3.trainable.processes.StandardClassification method*), 44
freeze_features() (*dn3.trainable.models.Classifier method*), 32
from_dataset() (*dn3.trainable.models.Classifier class method*), 32

G

get_label_balance() (*in module dn3.trainable.processes*), 44
get_sessions() (*dn3.data.dataset.Dataset method*), 26
get_targets() (*dn3.data.dataset.Dataset method*), 26
get_targets() (*dn3.data.dataset.Thinker method*), 29
get_thinkers() (*dn3.data.dataset.Dataset method*), 26
get_transform() (*dn3.transforms.preprocessors.Preprocessor method*), 46

I

IndexSelect (*class in dn3.trainable.layers*), 36

L

LDAMLoss (*class in dn3.trainable.processes*), 42
lmso() (*dn3.data.dataset.Dataset method*), 26
load_best() (*dn3.trainable.processes.BaseProcess method*), 42
LogRegNetwork (*class in dn3.trainable.models*), 33
loso() (*dn3.data.dataset.Dataset method*), 26

M

make_new_classification_layer() (*dn3.trainable.models.Classifier method*), 32
make_new_classification_layer() (*dn3.trainable.models.StrideClassifier method*), 33
MappingDeep1010 (*class in dn3.transforms.instance*), 45
module
    dn3.configuratron.config, 19
    dn3.data.dataset, 23
    dn3.trainable.layers, 34
    dn3.trainable.models, 31
    dn3.trainable.processes, 39
    dn3.transforms.batch, 46
    dn3.transforms.instance, 45
    dn3.transforms.preprocessors, 46

N

new_channels() (*dn3.transforms.instance.MappingDeep1010 method*), 45
NoisyBlankDeep1010 (*class in dn3.transforms.instance*), 46

P

parameters() (*dn3.trainable.processes.BaseProcess method*), 42

Permute (*class in dn3.trainable.layers*), 36
predict() (*dn3.trainable.processes.BaseProcess method*), 42
preprocess() (*dn3.configuratron.config.RawOnTheFlyRecording method*), 21
preprocess() (*dn3.data.dataset.Dataset method*), 27
preprocess() (*dn3.data.dataset.DN3ataset method*), 24
preprocess() (*dn3.data.dataset.EpochTorchRecording method*), 28
preprocess() (*dn3.data.dataset.RawTorchRecording method*), 28
preprocess() (*dn3.data.dataset.Thinker method*), 29
Preprocessor (*class in dn3.transforms.preprocessors*), 46

R

RawOnTheFlyRecording (*class in dn3.configuratron.config*), 21
RawTorchRecording (*class in dn3.data.dataset*), 28

S

safe_mode() (*dn3.data.dataset.Dataset method*), 27
same_channel_sets() (*in module dn3.transforms.instance*), 46
save_best() (*dn3.trainable.processes.BaseProcess method*), 42
scan_toplevel() (*dn3.configuratron.config.DatasetConfig method*), 21
sequence_length() (*dn3.data.dataset.Dataset property*), 27
sequence_length() (*dn3.data.dataset.DN3ataset property*), 24
sequence_length() (*dn3.data.dataset.Thinker property*), 30
set_scheduler() (*dn3.trainable.processes.BaseProcess method*), 42
sfreq() (*dn3.data.dataset.Dataset property*), 27
sfreq() (*dn3.data.dataset.DN3ataset property*), 24
sfreq() (*dn3.data.dataset.Thinker property*), 30
SpatialFilter (*class in dn3.trainable.layers*), 37
split() (*dn3.data.dataset.Thinker method*), 30
Squeeze (*class in dn3.trainable.layers*), 37
StandardClassification (*class in dn3.trainable.processes*), 43
StrideClassifier (*class in dn3.trainable.models*), 33

T

TemporalFilter (*class in dn3.trainable.layers*), 37
Thinker (*class in dn3.data.dataset*), 29
TIDNet (*class in dn3.trainable.models*), 33
to_numpy() (*dn3.data.dataset.DN3ataset method*), 24

U

update_id_returns() (*dn3.data.dataset.Dataset method*), 27

Z

ZScore (*class in dn3.transforms.instance*), 46