hypnettorch

Release 1.0

Christian Henning, Maria R. Cervera

Sep 07, 2023

CONTENTS:

CHAPTER

ONE

CUSTOM DATA HANDLERS FOR COMMON ML DATASETS

Contents

This folder contains data loaders for common datasets. Note, the code in this folder is a derivative of the dataloaders developed in this repo. For examples on how to use these data loaders with Tensorflow checkout the original code.

All dataloaders are derived from the abstract base class `hypnettorch.data.dataset.Dataset` to provide a common API to the user.

1.1 Preparation of datasets

Datasets not mentioned in this section will be automatically downloaded and processed.

Here you can find instructions about how to prepare some of the datasets for automatic processing.

1.1.1 Large-scale CelebFaces Attributes (CelebA) Dataset

CelebA is a dataset with over 200K celebrity images. It can be downloaded from here.

Google Drive will split the dataset into multiple zip-files. In the following, we explain, how you can extract these files on Linux. To decompress the sharded zip files, simply open a terminal, move to the downloaded zip-files and enter:

```
$ unzip '*.zip'
```

This will create a local folder named `CelebA`.

Afterwards move into the `Img` subfolder:

```
$ cd ./CelebA/Img/
```

You can now decide, whether you want to use the JPG or PNG encoded images.

For the jpeg images, you have to enter:

```
$ unzip img_align_celeba.zip
```

This will create a folder `img_align_celeba`, containing all images in jpeg format.

To save space on your local machine, you may delete the zip file via `rm img_align_celeba.zip`.

The same images are also available in png format. To extract these, you have to move in the corresponding subdirectory via `cd img_align_celeba_png.7z`. You can now extract the sharded 7z files by entering:

```
$ 7z e img_align_celeba_png.7z.001
```

Again, you may now delete the archives to save space via `rm img_align_celeba_png.7z.0*`.

You can proceed similarly if you want to work with the original images located in the folder `img_celeba.7z`.

FYI, there are scripts available (e.g., here), that can be used to download the dataset.

1.1.2 Imagenet Large Scale Visual Recognition Challenge 2012 (ILSVRC2012)

The ILSVRC2012 dataset is a subset of the ImageNet dataset and contains over 1.2 Mio. training images depicting natural image scenes of 1,000 object classes. The dataset can be downloaded here here.

For our program to be able to use the dataset, it has to be prepared as described here.

In the following, we recapitulate the required steps (which are executed from the directory in which the dataset has been loaded to).

1. Download the training and validation images as well as the development kit for task 1 & 2.
2. Extract the training data.

```
mkdir train && mv ILSVRC2012_img_train.tar train/ && cd train
tar -xvf ILSVRC2012_img_train.tar && rm -f ILSVRC2012_img_train.tar
find . -name "*.tar" | while read NAME ; do mkdir -p "${NAME%.tar}"; tar -xvf "$
↪{NAME}" -C "${NAME%.tar}"; rm -f "${NAME}"; done
cd ..
```

Note, this step deletes the the downloaded tar-file. If this behavior is not desired replace the command `rm -f ILSVRC2012_img_train.tar` with `mv ILSVRC2012_img_train.tar ...`

3. Extract the validation data and move images to subfolders.

```
mkdir val && mv ILSVRC2012_img_val.tar val/ && cd val && tar -xvf ILSVRC2012_img_
↪val.tar
wget -qO- https://raw.githubusercontent.com/soumith/imagenetloader.torch/master/
↪valprep.sh | bash
cd ..
```

This step ensures that the validation samples are grouped in the same folder structure as the training samples, i.e., validation images are stored under their corresponding WordNet ID (*WNID*).

4. Extract the meta data:

```
mkdir meta && mv ILSVRC2012_devkit_t12.tar.gz meta/ && cd meta && tar -xvzf␣
↪ILSVRC2012_devkit_t12.tar.gz --strip 1
cd ..
```

1.1.3 Udacity Steering Angle Prediction

The CH2 steering angle prediction dataset from Udacity can be downloaded here. In the following, we quickly explain how we expect the downloads to be preprocessed for our datahandler to work.

You may first decompress the files, after which you should have two subfolders `Ch2_001` (for the test data) and ``Ch2_002` (for the training data). You may replace the file `Ch2_001/HMB_3_release.bag` with the complete test set `Ch2_001/HMB_3.bag`.

We use this docker tool to extract the information from the Bag files and align the steering information with the recorded images.

Simply clone the repository and execute the `./build.sh`. This issue helped us to overcome an error during the build.

Afterwards, the bagfiles can be extracted using (note, that in- and output directory must be specified using absolute paths), for instance

```
sudo ./run-bagdump.sh -i /data/udacity/Ch2_001/ -o /data/udacity/Ch2_001/
```

and

```
sudo ./run-bagdump.sh -i /data/udacity/Ch2_002/ -o /data/udacity/Ch2_002/
```

The data handler only requires the `center/` folder and the file `interpolated.csv`. All remaining extracted data (for instance, left and right camera images) can be deleted.

Alternatively, the dataset can be downloaded from here. This dataset appears to contain images recorded a month before the official Challenge 2 dataset was recorded. We could not find any information whether the experimental conditions are identical (e.g., whether steering angles are directly comparable). Additionally, the dataset appears to contain situations like parking, where the vehicle doesn't move and there is no road ahead. Anyway, if desired, the

dataset can be processed similarly to the above mentioned. One may first want to filter the bag file, to only keep information relevant for the task at hand, e.g.:

```
rosbag filter dataset-2-2.bag dataset-2-2_filtered.bag "topic == '/center_camera/image_
↪color' or  topic == '/vehicle/steering_report'"
```

The bag file can be extracted in to `center/` folder and a file `interpolated.csv` as described above, using `./run-bagdump.sh`.

1.2 API

1.2.1 Dataset Interface

The module `data.dataset` contains a template for a dataset interface, that can be used to feed data into neural networks.

The implementation is based on an earlier implementation of a class I used in another project:

> https://git.io/fN1a6

At the moment, the class holds all data in memory and is therefore not meant for bigger datasets. Though, it is easy to design wrappers that overcome this limitation (e.g., see abstract base class `data.large_img_dataset.LargeImgDataset`).

hypnettorch.data.dataset.Dataset.get_test_ids()	Get unique identifiers all test samples.
hypnettorch.data.dataset.Dataset.get_train_ids()	Get unique identifiers all training samples.
hypnettorch.data.dataset.Dataset.get_val_ids()	Get unique identifiers all validation samples.
hypnettorch.data.dataset.Dataset.get_test_inputs()	Get the inputs of all test samples.
hypnettorch.data.dataset.Dataset.get_test_outputs([...])	Get the outputs (targets) of all test samples.
hypnettorch.data.dataset.Dataset.get_train_inputs()	Get the inputs of all training samples.
hypnettorch.data.dataset.Dataset.get_train_outputs([...])	Get the outputs (targets) of all training samples.
hypnettorch.data.dataset.Dataset.get_val_inputs()	Get the inputs of all validation samples.
hypnettorch.data.dataset.Dataset.get_val_outputs([...])	Get the outputs (targets) of all validation samples.
hypnettorch.data.dataset.Dataset.input_to_torch_tensor(x, ...)	This method can be used to map the internal numpy arrays to PyTorch tensors.
hypnettorch.data.dataset.Dataset.is_image_dataset()	Are input (resp.
hypnettorch.data.dataset.Dataset.next_test_batch(...)	Return the next random test batch.
hypnettorch.data.dataset.Dataset.next_train_batch(...)	Return the next random training batch.
hypnettorch.data.dataset.Dataset.next_val_batch(...)	Return the next random validation batch.
hypnettorch.data.dataset.Dataset.test_iterator(...)	A generator to loop over the test set.
hypnettorch.data.dataset.Dataset.train_iterator(...)	A generator to loop over the training set.
hypnettorch.data.dataset.Dataset.val_iterator(...)	A generator to loop over the validation set.
hypnettorch.data.dataset.Dataset.output_to_torch_tensor(y, ...)	Similar to method `input_to_torch_tensor()`, just for dataset outputs.
hypnettorch.data.dataset.Dataset.plot_samples(...)	Plot samples belonging to this dataset.
hypnettorch.data.dataset.Dataset.reset_batch_generator([...])	The batch generation possesses a memory.
hypnettorch.data.dataset.Dataset.tf_input_map([mode])	This method should be used by the map function of the Tensorflow Dataset interface (`tf.data.Dataset.map`).
hypnettorch.data.dataset.Dataset.tf_output_map([mode])	Similar to method `tf_input_map()`, just for dataset outputs.
hypnettorch.data.dataset.Dataset.test_ids_to_indices(...)	Translate an array of test sample identifiers to test indices.
hypnettorch.data.dataset.Dataset.train_ids_to_indices(...)	Translate an array of training sample identifiers to training indices.
hypnettorch.data.dataset.Dataset.val_ids_to_indices(...)	Translate an array of validation sample identifiers to validation indices.

class hypnettorch.data.dataset.**Dataset**

Bases: ABC

A general dataset template that can be used as a simple and consistent interface. Note, that this is an abstract class that should not be instantiated.

In order to write an interface for another dataset, you have to implement an inherited class. You must always call the constructor of this base class first when instantiating the implemented subclass.

Note, the internals are stored in the private member `_data`, that is described in the constructor.

property classification

Whether the dataset is a classification or regression dataset.

Type
bool

abstract get_identifier()

Returns the name of the dataset.

Returns
The dataset its (unique) identifier.

Return type
(str)

get_test_ids()

Get unique identifiers all test samples.

See documentation of method *get_train_ids()* for details.

Returns
A 1D numpy array.

Return type
(numpy.ndarray)

get_test_inputs()

Get the inputs of all test samples.

See documentation of method *get_train_inputs()* for details.

Returns
A 2D numpy array.

Return type
(numpy.ndarray)

get_test_outputs(*use_one_hot=None*)

Get the outputs (targets) of all test samples.

See documentation of method *get_train_outputs()* for details.

Parameters
(....) – See docstring of method *get_train_outputs()*.

Returns
A 2D numpy array.

Return type
(numpy.ndarray)

`get_train_ids()`

Get unique identifiers all training samples.

Each sample in the dataset has a unique identifier (independent of the dataset split it is assigned to).

Note: Sample identifiers do not correspond to the indices of samples within a dataset split (i.e., the returned identifiers of this method cannot be used as indices for the returned arrays of methods *get_train_inputs()* and *get_train_outputs()*)

Returns
A 1D numpy array containing the unique identifiers for all training samples.

Return type
(numpy.ndarray)

`get_train_inputs()`

Get the inputs of all training samples.

Note, that each sample is encoded as a single vector. One may use the attribute *in_shape* to decode the actual shape of an input sample.

Returns
A 2D numpy array, where each row encodes a training sample.

Return type
(numpy.ndarray)

`get_train_outputs(use_one_hot=None)`

Get the outputs (targets) of all training samples.

Note, that each sample is encoded as a single vector. One may use the attribute *out_shape* to decode the actual shape of an output sample. Keep in mind, that classification samples might be one-hot encoded.

Parameters
use_one_hot (*bool*) – For classification samples, the encoding of the returned samples can be either "one-hot" or "class index". This option is ignored for datasets other than classification sets. If `None`, the dataset its default encoding is returned.

Returns
A 2D numpy array, where each row encodes a training target.

Return type
(numpy.ndarray)

`get_val_ids()`

Get unique identifiers all validation samples.

See documentation of method *get_train_ids()* for details.

Returns
A 1D numpy array. Returns `None` if no validation set exists.

Return type
(numpy.ndarray)

`get_val_inputs()`

Get the inputs of all validation samples.

See documentation of method *get_train_inputs()* for details.

Returns
A 2D numpy array. Returns `None` if no validation set exists.

Return type
(numpy.ndarray)

`get_val_outputs`(*use_one_hot=None*)

Get the outputs (targets) of all validation samples.

See documentation of method *get_train_outputs()* for details.

Parameters
(....) – See docstring of method *get_train_outputs()*.

Returns
A 2D numpy array. Returns `None` if no validation set exists.

Return type
(numpy.ndarray)

`property in_shape`

The original shape of an input sample.

Note, that samples are encoded by this class as individual vectors (e.g., an MNIST sample is ancoded as 784 dimensional vector, but its original shape is: `[28, 28, 1]`). A sequential sample is encoded by concatenating all timeframes. Hence, the number of timesteps can be decoded by dividing a single sample vector by `np.prod(in_shape)`.

Type
list

`input_to_torch_tensor`(*x*, *device*, *mode='inference'*, *force_no_preprocessing=False*, *sample_ids=None*)

This method can be used to map the internal numpy arrays to PyTorch tensors.

Note, subclasses might overwrite this method and add data preprocessing/ augmentation.

Parameters

- **x** (*numpy.ndarray*) – A 2D numpy array, containing inputs as provided by this dataset.
- **device** (*torch.device or int*) – The PyTorch device onto which the input should be mapped.
- **mode** (*str*) – See docstring of method *tf_input_map()*. Valid values are: `train` and `inference`.
- **force_no_preprocessing** (*bool*) – In case preprocessing is applied to the inputs (e.g., normalization or random flips/crops), this option can be used to prohibit any kind of manipulation. Hence, the inputs are transformed into PyTorch tensors on an "as is" basis.
- **sample_ids** (*numpy.ndarray*) – See method *train_ids_to_indices()*. Instantiation of this class might make use of this information, for instance in order to reduce the amount of zero padding within a mini-batch.

Returns
The given input `x` as PyTorch tensor.

Return type
(torch.Tensor)

`is_image_dataset`()

Are input (resp. output) samples images?

Note, for sequence datasets, this method just returns whether a single frame encodes an image.

Returns

Tuple containing two booleans:

- **input_is_img**
- **output_is_img**

Return type
(tuple)

`property is_one_hot`

Whether output labels are one-hot encoded for a classification task (`None` otherwise).

Type
bool or None

`next_test_batch`(*batch_size*, *use_one_hot=None*, *return_ids=False*)

Return the next random test batch.

See documentation of method *next_train_batch()* for details.

Parameters
(....) – See docstring of method *next_train_batch()*.

Returns

List containing the following 2D numpy arrays:

- **batch_inputs**
- **batch_outputs**
- **batch_ids** (optional)

Return type
(list)

`next_train_batch`(*batch_size*, *use_one_hot=None*, *return_ids=False*)

Return the next random training batch.

If the behavior of this method should be reproducible, please define a numpy random seed.

Parameters

- **(....)** – See docstring of method *get_train_outputs()*.
- **batch_size** (*int*) – The size of the returned batch.
- **return_ids** (*bool*) – If `True`, a third value will be returned that is a 1D numpy array containing sample identifiers.

  Note: Those integer values are internal unique sample identifiers and in general **do not** correspond to indices within the corresponding dataset split (i.e., the training split in this case).

Returns

List containing the following 2D numpy arrays:

- **batch_inputs**: The inputs of the samples belonging to the batch.
- **batch_outputs**: The outputs of the samples belonging to the batch.
- **batch_ids** (optional): See option `return_ident`.

Return type
(list)

next_val_batch(*batch_size*, *use_one_hot=None*, *return_ids=False*)

Return the next random validation batch.

See documentation of method *next_train_batch()* for details.

Parameters
(....) – See docstring of method *next_train_batch()*.

Returns

List containing the following 2D numpy arrays:

- **batch_inputs**
- **batch_outputs**
- **batch_ids** (optional)

Returns None if no validation set exists.

Return type
(list)

property num_classes

The number of classes for a classification task (None otherwise).

Type
int or None

property num_test_samples

The number of test samples.

Type
int

property num_train_samples

The number of training samples.

Type
int

property num_val_samples

The number of validation samples.

Type
int

property out_shape

The original shape of an output sample (see *in_shape*).

Type
list

output_to_torch_tensor(*y*, *device*, *mode='inference'*, *force_no_preprocessing=False*, *sample_ids=None*)

Similar to method *input_to_torch_tensor()*, just for dataset outputs.

Note, in this default implementation, it is also does not perform any data preprocessing.

Parameters
(....) – See docstring of method *input_to_torch_tensor()*.

Returns
The given output `y` as PyTorch tensor.

Return type
(torch.Tensor)

plot_samples(*title*, *inputs*, *outputs=None*, *predictions=None*, *num_samples_per_row=4*, *show=True*, *filename=None*, *interactive=False*, *figsize=(10, 6)*, ***kwargs*)

Plot samples belonging to this dataset. Each sample will be plotted in its own subplot.

Parameters

- **title** (*str*) – The title of the whole figure.
- **inputs** (*numpy.ndarray*) – A 2D numpy array, where each row is an input sample.
- **outputs** (*numpy.ndarray, optional*) – A 2D numpy array of actual dataset targets.
- **predictions** (*numpy.ndarray, optional*) – A 2D numpy array of predicted output samples (i.e., output predicted by a neural network).
- **num_samples_per_row** (*int*) – Maximum number of samples plotted per row in the generated figure.
- **show** (*bool*) – Whether the plot should be shown.
- **filename** (*str, optional*) – If provided, the figure will be stored under this filename.
- **interactive** (*bool*) – Turn on interactive mode. We mainly use this option to ensure that the program will run in background while figure is displayed. The figure will be displayed until another one is displayed, the user closes it or the program has terminated. If this option is deactivated, the program will freeze until the user closes the figure. Note, if using the iPython inline backend, this option has no effect.
- **figsize** (*tuple*) – A tuple, determining the size of the figure in inches.
- ****kwargs** (*optional*) – Optional keyword arguments that can be dataset dependent.

reset_batch_generator(*train=True*, *test=True*, *val=True*)

The batch generation possesses a memory. Hence, the samples returned depend on how many samples already have been retrieved via the next- batch functions (e.g., *next_train_batch()*). This method can be used to reset these generators.

Parameters

- **train** (*bool*) – If `True`, the generator for *next_train_batch()* is reset.
- **test** (*bool*) – If `True`, the generator for *next_test_batch()* is reset.
- **val** (*bool*) – If `True`, the generator for *next_val_batch()* is reset, if a validation set exists.

property sequence

Whether the dataset contains sequences (samples have temporal structure). In case of a sequential dataset, the temporal structure can be decoded via the shape attributes of in- and outputs. Note, that all samples are internally zero-padded to the same length.

Type
bool

property shuffle_test_samples

Whether the method *next_test_batch()* returns test samples in random order at every epoch. Defaults to `True`, i.e., samples have a random ordering every epoch.

Type
bool

Setter
Note, setting this attribute will reset the current batch generator, such that the next call to the method *next_test_batch()* results in starting a sweep through a new epoch (full batch).

property shuffle_val_samples

Same as *shuffle_test_samples* for samples from the validation set.

Type
bool

test_ids_to_indices(*sample_ids*)

Translate an array of test sample identifiers to test indices.

See documentation of method *train_ids_to_indices()* for details.

Parameters
(....) – See docstring of method *train_ids_to_indices()*.

Returns
A 1D numpy array.

Return type
(numpy.ndarray)

test_iterator(*batch_size*, *return_remainder=True*, ***kwargs*)

A generator to loop over the test set.

See documentation of method *train_iterator()*.

Parameters
(....) – See docstring of method *train_iterator()*.

Yields
(list) – The same list that would be returned by method *next_test_batch()* but additionally prepended with the batch size.

tf_input_map(*mode='inference'*)

This method should be used by the map function of the Tensorflow Dataset interface (`tf.data.Dataset.map`). In the default case, this is just an identity map, as the data is already in memory.

There might be cases, in which the full dataset is too large for the working memory, and therefore the data currently needed by Tensorflow has to be loaded from disk. This function should be used as an interface for this process.

Parameters
mode (*str*) – Is the data needed for training or inference? This distinction is important, as it might change the way the data is processed (e.g., special random data augmentation might apply during training but not during inference. The parameter is a string with the valid values being `train` and `inference`.

Returns
A function handle, that maps the given input tensor to the preprocessed input tensor.

Return type
(function)

tf_output_map(*mode='inference'*)

Similar to method *tf_input_map()*, just for dataset outputs.

Note, in this default implementation, it is also just an identity map.

Parameters
(....) – See docstring of method *tf_input_map()*.

Returns
A function handle.

Return type
(function)

train_ids_to_indices(*sample_ids*)

Translate an array of training sample identifiers to training indices.

This method translates unique training identifiers (see method *get_train_ids()*) to actual training indices, that can be used to index the training set.

Parameters
sample_ids (*numpy.ndarray*) – 1D numpy array of unique sample IDs (e.g., those returned when using option `return_ids` of method *next_train_batch()*).

Returns
A 1D array of training indices that has the same length as `sample_ids`.

Return type
(numpy.ndarray)

train_iterator(*batch_size*, *return_remainder=True*, ***kwargs*)

A generator to loop over the training set.

This generator yields the return value of *next_train_batch()* prepended with the current batch size.

Example

```
for batch_size, x, y in data.train_iterator(32):
    x_t = data.input_to_torch_tensor(x, device, mode='train')
    y_t = data.output_to_torch_tensor(y, device, mode='train')

    # ...
```

```
for batch_size, x, y, ids in data.train_iterator(32, \
        return_ids=True):
    x_t = data.input_to_torch_tensor(x, device, mode='train')
    y_t = data.output_to_torch_tensor(y, device, mode='train')

    # ...
```

Note: This method will only temporarily modify the internal batch generator (see method *reset_batch_generator()*) until the epoch is completed.

Parameters

- **batch_size** (*int*) – The batch size used.

  Note: If `batch_size` is not an integer divider of *num_train_samples*, then the last yielded batch will be smaller if `return_remainder` is `True`.

- **return_remainder** (*bool*) – The last batch might have to be smaller if `batch_size` is not an integer divider of *num_train_samples*. If this attribute is `False`, this last part is not yielded and all batches have the same size.

  Note: If `return_remainder` is se tto `False`, then it may be that not all training samples are yielded.

- ****kwargs** – Keyword arguments that are passed to method *next_train_batch()*.

Yields
(list) – The same list that would be returned by method *next_train_batch()* but additionally prepended with the batch size.

val_ids_to_indices(*sample_ids*)

Translate an array of validation sample identifiers to validation indices.

See documentation of method *train_ids_to_indices()* for details.

Parameters
(....) – See docstring of method *train_ids_to_indices()*.

Returns
A 1D numpy array.

Return type
(numpy.ndarray)

val_iterator(*batch_size*, *return_remainder=True*, ***kwargs*)

A generator to loop over the validation set.

See documentation of method *train_iterator()*.

Parameters
(....) – See docstring of method *train_iterator()*.

Yields
(list) – The same list that would be returned by method *next_val_batch()* but additionally prepended with the batch size.

1.2.2 Wrapper for large image datasets

The module `data.large_img_dataset` contains an abstract wrapper for large datasets, that have images as inputs. Typically, these datasets are too large to be loaded into memory. Though, their outputs (labels) can still easily be hold in memory. Hence, the idea is, that instead of loading the actual images, we load the paths for each image into memory. Then we can load the images from disk as needed.

To sum up, handlers that implement this interface will hold the outputs and paths for the input images of the whole dataset in memory, but not the actual images.

As an alternative, one can implement wrappers for HDF5 and TFRecord files.

Here is a simple example that illustrates the format of the dataset:

https://www.tensorflow.org/guide/datasets#decoding_image_data_and_resizing_it

In case of working with PyTorch, rather than using the internal methods for batch processing (such as `data.dataset.Dataset.next_train_batch()`) one should adapt PyTorch its data processing utilities (consisting of `torch.utils.data.Dataset` and `torch.utils.data.DataLoader`) in combination with class attributes such as `data.large_img_dataset.LargeImgDataset.torch_train`.

class hypnettorch.data.large_img_dataset.**LargeImgDataset**(*imgs_path*, *png_format=False*)

Bases: *Dataset*

A general dataset template for datasets with images as inputs, that are locally stored as individual files. Note, that this is an abstract class that should not be instantiated.

Hints, when implementing the interface:

- Attribute `data.dataset.Dataset.in_shape` still has to be correctly implemented, independent of the fact, that the actual input data is a list of strings.

Parameters

- **imgs_path** (*str*) – The path to the folder, containing the image files (the actual image paths contained in the input data (see e.g., `data.dataset.Dataset.get_train_inputs()`) will be concatenated to this path).
- **png_format** (*bool*) – The images are typically assumed to be jpeg encoded. You may change this to png enocded images.

get_test_inputs()

Get the inputs of all test samples.

Returns
An np.chararray, where each row corresponds to an image file name.

Return type
(numpy.chararray)

get_train_inputs()

Get the inputs of all training samples.

Returns
An np.chararray, where each row corresponds to an image file name.

Return type
(numpy.chararray)

get_val_inputs()

Get the inputs of all validation samples.

Returns
An np.chararray, where each row corresponds to an image file name. If no validation set exists, `None` will be returned.

Return type
(numpy.chararray)

property imgs_path

The base path of all images.

Type
str

input_to_torch_tensor(*x*, *device*, *mode='inference'*, *force_no_preprocessing=False*, *sample_ids=None*)

Note, this method has been overwritten from the base class. It should not be used for large image datasets. Instead, the class should provide instances of class `torch.utils.data.Dataset` for training, validation and test set:

- *torch_train*
- *torch_test*
- *torch_val*

property png_format_used

Whether png or jped encoding of images is assumed.

Type
bool

read_images(*inputs*)

For the given filenames, read and return the images.

Parameters
inputs (*numpy.chararray*) – An np.chararray of filenames.

Returns
A 2D numpy array, where each row contains a picture.

Return type
(numpy.ndarray)

tf_input_map(*mode='inference'*)

Note, this method has been overwritten from the base class.

It provides a function handle that loads images from file, resizes them to match the internal input image size and then flattens the image to a vector.

Parameters
(....) – See docstring of method `data.dataset.Dataset.tf_input_map()`.

Returns
A function handle, that maps the given input tensor to the preprocessed input tensor.

Return type
(function)

property torch_test

The PyTorch compatible test dataset.

Type
torch.utils.data.Dataset

property torch_train

The PyTorch compatible training dataset.

Type
torch.utils.data.Dataset

property torch_val

The PyTorch compatible validation dataset.

Type
torch.utils.data.Dataset

1.2.3 Wrapper for sequential datasets

The module `data.sequential_dataset` contains an abstract wrapper for datasets containing sequential data.

Even though the dataset interface `data.dataset.Dataset` contains basic support for sequential datasets, this wrapper was considered necessary to increase the convinience when working with sequential datasets (especially, if those datasets contain sequences of varying lengths).

class hypnettorch.data.sequential_dataset.**SequentialDataset**

Bases: *Dataset*

A general wrapper for datasets with sequential inputs and outpus.

get_in_seq_lengths(*sample_ids*)

Get the unpadded input sequence lengths for given samples.

Parameters
sample_ids (*numpy.ndarray*) – A 1D numpy array of unique sample identifiers. Please see documentation of option `return_ids` of method `data.dataset.Dataset.next_train_batch()` as well as method `data.dataset.Dataset.get_train_ids()` for more information of sample identifiers.

Returns
A 1D array of the same length as `sample_ids` containing the unpadded input sequence lengths of these samples.

Return type
(numpy.ndarray)

get_out_seq_lengths(*sample_ids*)

Get the unpadded output sequence lengths for given samples.

See documentation of method *get_in_seq_lengths()*.

Parameters
(....) – See docstring of method *get_in_seq_lengths()*.

Returns
A 1D numpy array.

Return type
(numpy.ndarray)

input_to_torch_tensor(*x*, *device*, *mode='inference'*, *force_no_preprocessing=False*, *sample_ids=None*)

This method can be used to map the internal numpy arrays to PyTorch tensors.

Parameters
(....) – See docstring of method `data.dataset.Dataset.input_to_torch_tensor()`.

Returns
The given input `x` as PyTorch tensor. It has dimensions `[T, B, *in_shape]`, where `T` is the number of time steps (see attribute *max_num_ts_in*), `B` is the batch size and `in_shape` refers to the input feature shape, see `data.dataset.Dataset.in_shape`.

Return type
(torch.Tensor)

property max_num_ts_in

The maximum number of timesteps input sequences may have.

Note: Internally, all input sequences are stored according to this length using zero-padding.

Type
int

property max_num_ts_out

The maximum number of timesteps output sequences may have.

Note: Internally, all input sequences are stored according to this length using zero-padding.

Type
int

output_to_torch_tensor(*y*, *device*, *mode='inference'*, *force_no_preprocessing=False*, *sample_ids=None*)

Similar to method *input_to_torch_tensor()*, just for dataset outputs.

Parameters
(....) – See docstring of method `data.dataset.Dataset.output_to_torch_tensor()`.

Returns
The given input `x` as PyTorch tensor. It has dimensions `[T, B, *out_shape]`, where `T` is the number of time steps (see attribute *max_num_ts_out*), `B` is the batch size and `out_shape` refers to the output feature shape, see `data.dataset.Dataset.out_shape`.

Return type
(torch.Tensor)

1.2.4 CelebA Dataset

The module `data.celeba_data` contains a handler for the CelebA dataset.

More information about the dataset can be retrieved from:
http://mmlab.ie.cuhk.edu.hk/projects/CelebA.html

Note, in the current implementation, this handler will not download and extract the dataset for you. You have to do this manually by following the instructions of the README file (which is located in the same folder as this file).

Note, this dataset has not yet been prepared for PyTorch use!

class hypnettorch.data.celeba_data.**CelebAData**(*data_path*, *use_png=False*, *shape=None*)

Bases: *LargeImgDataset*

An instance of the class shall represent the CelebA dataset.

The input data of the dataset will be strings to image files. The output data will be vectors of booleans, denoting whether a certain type of attribute is present in the picture.

Note: The dataset has to be already downloaded and extracted before this class can be instantiated. See the local README file for details.

Parameters

- **data_path** (*str*) – Where should the dataset be read from?
- **use_png** (*bool*) – Whether the png rather than the jpeg images should be used. Note, this class only considers the aligned and cropped images.
- **shape** (*optional*) – If given, this images loaded from disk will be reshaped to that shape.

`get_attribute_names()`

Get the names of the different attributes classified by this dataset.

Returns
A list of attributes. The order of the list will have the same order as the output labels.

Return type
(list)

`get_identifier()`

Returns the name of the dataset.

1.2.5 CIFAR-10 Dataset

The module `data.cifar10_data` contains a handler for the CIFAR 10 dataset.

The dataset consists of 60000 32x32 colour images in 10 classes, with 6000 images per class. There are 50000 training images and 10000 test images.

Information about the dataset can be retrieved from:
https://www.cs.toronto.edu/~kriz/cifar.html

`class hypnettorch.data.cifar10_data.CIFAR10Data(`*data_path*, *use_one_hot=False*, *use_data_augmentation=False*, *validation_size=5000*, *use_cutout=False*`)`

Bases: *Dataset*

An instance of the class shall represent the CIFAR-10 dataset.

Note, the constructor does not safe a data dump (via pickle) as, for instance, the MNIST data handler (`data.mnist_data.MNISTData`) does. The reason is, that the downloaded files are already in a nice to read format, such that the time saved to read the file from a dump file is minimal.

Note: By default, input samples are provided in a range of `[0, 1]`.

Parameters

- **data_path** (*str*) – Where should the dataset be read from? If not existing, the dataset will be downloaded into this folder.
- **use_one_hot** (*bool*) – Whether the class labels should be represented in a one-hot encoding.
- **use_data_augmentation** (*bool*) – Note, this option currently only applies to input batches that are transformed using the class member *input_to_torch_tensor()* (hence, **only available for PyTorch**, so far).

Note: If activated, the statistics of test samples are changed as a normalization is applied.

- **validation_size** (*int*) – The number of validation samples. Validation samples will be taking from the training set (the first n samples).
- **use_cutout** (*bool*) – Whether option `apply_cutout` should be set of method *torch_input_transforms()*. We use cutouts of size `16 x 16` as recommended here.

  Note: Only applies if `use_data_augmentation` is set.

get_identifier()

Returns the name of the dataset.

input_to_torch_tensor(*x*, *device*, *mode='inference'*, *force_no_preprocessing=False*, *sample_ids=None*)

This method can be used to map the internal numpy arrays to PyTorch tensors.

Note, this method has been overwritten from the base class.

The input images are preprocessed if data augmentation is enabled. Preprocessing involves normalization and (for training mode) random perturbations.

Parameters

(....) – See docstring of method `data.dataset.Dataset.input_to_torch_tensor()`.

Returns

The given input `x` as PyTorch tensor.

Return type

(torch.Tensor)

plot_sample(*image*, *label=None*, *figsize=1.5*, *interactive=False*, *file_name=None*)

Plot a single CIFAR-10 sample.

This method is thought to be helpful for evaluation and debugging purposes.

Deprecated since version 1.0: Please use method `data.dataset.Dataset.plot_samples()` instead.

Parameters

- **image** – A single CIFAR-10 image (given as 1D vector).
- **label** – The label of the given image.
- **figsize** – The height and width of the displayed image.
- **interactive** – Turn on interactive mode. Thus program will run in background while figure is displayed. The figure will be displayed until another one is displayed, the user closes it or the program has terminated. If this option is deactivated, the program will freeze until the user closes the figure.
- **file_name** – (optional) If a file name is provided, then the image will be written into a file instead of plotted to the screen.

static torch_augment_images(*x*, *device*, *transform*, *img_shape=[32, 32, 3]*)

Augment CIFAR-10 images using a given PyTorch transformation.

Parameters

- **x** (*numpy.ndarray*) – A 2D-Numpy array containing CIFAR-10 images.
- **device** (*torch.device or int*) – The PyTorch device on which the resulting tensor should be.

- **transform** – A `torchvision.transforms` method to modify the data.

Returns
The augmented images as PyTorch tensor.

Return type
(torch.Tensor)

static torch_input_transforms(*apply_rand_hflips=True*, *apply_cutout=False*, *cutout_length=16*, *cutout_n_holes=1*)

Get data augmentation pipelines for CIFAR-10 inputs.

Note, the augmentation is inspired by the augmentation proposed in:
https://www.aiworkbox.com/lessons/augment-the-cifar10-dataset-using-the-randomhorizontalflip-and-randomcrop-tra

---

Note: We use the same data augmentation pipeline for CIFAR-100, as the images are very similar. Here is an example where they use slightly different normalization values, but we ignore this for now: https://zhenye-na.github.io/2018/10/07/pytorch-resnet-cifar100.html

---

Parameters

- **apply_rand_hflips** (*bool*) – Apply random horizontal flips during training.
- **apply_cutout** (*bool*) – Whether the cutout transformation should be applied to training inputs (see class `utils.torch_utils.CutoutTransform`).
- **cutout_length** (*int*) – If `apply_cutout` is `True`, then this will be passed as constructor argument `length` to class `utils.torch_utils.CutoutTransform`.
- **cutout_n_holes** (*int*) – If `apply_cutout` is `True`, then this will be passed as constructor argument `n_holes` to class `utils.torch_utils.CutoutTransform`.

Returns

Tuple containing:

- **train_transform**: A transforms pipeline that applies random transformations and normalizes the image.
- **test_transform**: Similar to train_transform, but no random transformations are applied.

Return type
(tuple)

1.2.6 CIFAR-100 Dataset

The module `data.cifar100_data` contains a handler for the CIFAR 100 dataset.

The dataset consists of 60000 32x32 colour images in 100 classes, with 600 images per class. There are 50000 training images and 10000 test images.

Information about the dataset can be retrieved from:
https://www.cs.toronto.edu/~kriz/cifar.html

class hypnettorch.data.cifar100_data.**CIFAR100Data**(*data_path*, *use_one_hot=False*, *use_data_augmentation=False*, *validation_size=5000*, *use_cutout=False*)

Bases: *Dataset*

An instance of the class shall represent the CIFAR-100 dataset.

Parameters

- **data_path** (*str*) – Where should the dataset be read from? If not existing, the dataset will be downloaded into this folder.
- **use_one_hot** (*bool*) – Whether the class labels should be represented in a one-hot encoding.
- **use_data_augmentation** (*bool*) – Note, this option currently only applies to input batches that are transformed using the class member *input_to_torch_tensor()* (hence, **only available for PyTorch**, so far).

  Note: If activated, the statistics of test samples are changed as a normalization is applied (identical to the of class `data.cifar10_data.CIFAR10Data`).

- **validation_size** (*int*) – The number of validation samples. Validation samples will be taking from the training set (the first n samples).
- **use_cutout** (*bool*) – Whether option `apply_cutout` should be set of method `torch_input_transforms()`. We use cutouts of size `8 x 8` as recommended here.

  Note: Only applies if `use_data_augmentation` is set.

  ---

get_identifier()

Returns the name of the dataset.

input_to_torch_tensor(*x*, *device*, *mode='inference'*, *force_no_preprocessing=False*, *sample_ids=None*)

This method can be used to map the internal numpy arrays to PyTorch tensors.

Note, this method has been overwritten from the base class.

The input images are preprocessed if data augmentation is enabled. Preprocessing involves normalization and (for training mode) random perturbations.

Parameters

(....) – See docstring of method `data.dataset.Dataset.input_to_torch_tensor()`.

Returns

The given input `x` as PyTorch tensor.

Return type

(torch.Tensor)

1.2.7 CUB-200-2011 Dataset

The module `data.cub_200_2011_data` contains a dataloader for the Caltech-UCSD Birds-200-2011 Dataset (CUB-200-2011).

The dataset is available at:

> http://www.vision.caltech.edu/visipedia/CUB-200-2011.html

For more information on the dataset, please refer to the corresponding publication:

> Wah et al., "The Caltech-UCSD Birds-200-2011 Dataset", California Institute of Technology, 2011. http://www.vision.caltech.edu/visipedia/papers/CUB_200_2011.pdf

The dataset consists of 11,788 images divided into 200 categories. The dataset has a specified train/test split and a lot of additional information (bounding boxes, segmentation, parts annotation, ...) that we don't make use of yet.

Note: This dataset should not be confused with the older version CUB-200, containing only 6,033 images.

Note: We use the same data augmentation as for class `data.ilsvrc2012_data.ILSVRC2012Data`.

Note: The original category labels range from 1-200. We modify them to range from 0 - 199.

---

class hypnettorch.data.cub_200_2011_data.**CUB2002011**(*data_path*, *use_one_hot=False*, *num_val_per_class=0*)

Bases: *LargeImgDataset*

An instance of the class shall represent the CUB-200-2011 dataset.

The input data of the dataset will be strings to image files. The output data corresponds to object labels (bird categories).

Note: The dataset will be downloaded if not available.

Note: The original category labels range from 1-200. We modify them to range from 0 - 199.

Parameters

- **data_path** (*str*) – Where should the dataset be read from? If not existing, the dataset will be downloaded into this folder.
- **use_one_hot** (*bool*) – Whether the class labels should be represented in a one-hot encoding.

  Note: This option does not influence the internal PyTorch Dataset classes (e.g., cmp. `data.large_img_dataset.LargeImgDataset.torch_train`), that can be used in conjunction with PyTorch data loaders.

- **num_val_per_class** (*int*) – The number of validation samples per class. For instance: If value 10 is given, a validation set of size 5 * 200 = 1,000 is constructed (these samples will be removed from the training set).

  Note: Validation samples use the same data augmentation pipeline as test samples.

`get_identifier()`

Returns the name of the dataset.

`tf_input_map(`*mode='inference'*`)`

Not impemented.

1.2.8 Fashion-MNIST Dataset

The module `data.fashion_mnist` contains a handler for the Fashion-MNIST dataset.

The dataset was introduced in:

> Xiao et al., Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms, 2017.

This module contains a simple wrapper from the corresponding torchvision dataset to our dataset interface `data.dataset.Dataset`.

`class hypnettorch.data.fashion_mnist.FashionMNISTData(`*data_path*, *use_one_hot=False*, *validation_size=0*, *use_torch_augmentation=False*`)`

Bases: *Dataset*

An instance of the class shall represent the Fashion-MNIST dataset.

Note: By default, input samples are provided in a range of `[0, 1]`.

Parameters

- **data_path** (*str*) – Where should the dataset be read from? If not existing, the dataset will be downloaded into this folder.
- **use_one_hot** (*bool*) – Whether the class labels should be represented in a one-hot encoding.
- **validation_size** (*int*) – The number of validation samples. Validation samples will be taking from the training set (the first n samples).
- **use_torch_augmentation** (*bool*) – Apply data augmentation to inputs when calling method `data.dataset.Dataset.input_to_torch_tensor()`.

  The augmentation will be identical to the one provided by class `data.mnist_data.MNISTData`, **except** that during training also random horizontal flips are applied.

  Note: If activated, the statistics of test samples are changed as a normalization is applied.

`get_identifier`()

Returns the name of the dataset.

`input_to_torch_tensor`(*x*, *device*, *mode='inference'*, *force_no_preprocessing=False*, *sample_ids=None*)

This method can be used to map the internal numpy arrays to PyTorch tensors.

Note, this method has been overwritten from the base class.

If enabled via constructor option `use_torch_augmentation`, input images are preprocessed. Preprocessing involves normalization and (for training mode) random perturbations.

Parameters

`(....)` – See docstring of method `data.dataset.Dataset.input_to_torch_tensor()`.

Returns

The given input `x` as PyTorch tensor.

Return type

(torch.Tensor)

1.2.9 ILSVRC2012 Dataset

The module `data.ilsvrc2012_data` contains a handler for the Imagenet Large Scale Visual Recognition Challenge 2012 (ILSVRC2012) dataset, a subset of the ImageNet dataset:

http://www.image-net.org/challenges/LSVRC/2012/index

For more details on the dataset, please refer to:

Olga Russakovsky et al. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision 115*, no. 3 (December 1, 2015): 211–52, https://doi.org/10.1007/s11263-015-0816-y

---

Note: In the current implementation, this handler will not download and extract the dataset for you. You have to do this manually by following the instructions of the README file (which is located in the same folder as this file).

Note: We use the validation set as test set. A new (custom) validation set will be created by taking the first n samples from each training class as validation samples, where n is configured by the user.

Note: This dataset has not yet been prepared for Tensorflow use!

When using PyTorch, this class will create dataset classes (`torch.utils.data.Dataset`) for you for the training, testing and validation set. Afterwards, you can use these dataset instances to create data loaders:

```
train_loader = torch.utils.data.DataLoader(
    ilsvrc2012_data.torch_train, batch_size=256, shuffle=True,
    num_workers=4, pin_memory=True)
```

You should then use these Pytorch data loaders rather than class internal methods to work with the dataset.

PyTorch data augmentation is applied as defined by the method *ILSVRC2012Data.torch_input_transforms()*. Images will be resized and cropped to size 224 x 224.

class hypnettorch.data.ilsvrc2012_data.**ILSVRC2012Data**(*data_path*, *use_one_hot=False*, *num_val_per_class=0*)

Bases: *LargeImgDataset*

An instance of the class shall represent the ILSVRC2012 dataset.

The input data of the dataset will be strings to image files. The output data corresponds to object labels according to the `ILSVRC2012_ID` - 1.

Note: This is different from many other ILSVRC2012 data handlers, where the labels are computed based on the order of the training folder names (which correspond to WordNet IDs (`WNID`)).

Note: The dataset has to be already downloaded and extracted before this method can be called. See the local README file for details.

Parameters

- **data_path** (*str*) – Where should the dataset be read from? If not existing, the dataset will be downloaded into this folder.
- **use_one_hot** (*bool*) – Whether the class labels should be represented in a one-hot encoding. Note, class labels correspond to the `ILSVRC2012_ID` minus 1 (from 0 to 999).

  Note: This option does not influence the internal PyTorch Dataset classes (e.g., cmp. `data.large_img_dataset.LargeImgDataset.torch_train`), that can be used in conjunction with PyTorch data loaders.

- **num_val_per_class** (*int*) – The number of validation samples per class.

  Note: The actual ILSVRC2012 validation set is used as test set by this data handler. Therefore, a new validation set is constructed (if value greater than 0), using the same amount of samples per class. For instance: If value 50 is given, a validation set of size 50 * 1000 = 50,000 is constructed (these samples will be removed from the training set).

  Note: Validation samples use the same data augmentation pipeline as test samples.

get_identifier()

Returns the name of the dataset.

tf_input_map(*mode='inference'*)

Not impemented.

to_common_labels(*outputs*)

Translate between label conventions.

Translate a given set of labels (that correspond to the `ILSVRC2012_ID` (minus one) of their images) back to the labels provided by the `torchvision.datasets.ImageFolder` class.

Note: This would be the label convention for ImageNet used by PyTorch examples.

Parameters
outputs – Targets (as integers or 1-hot encodings).

Returns
The translated targets (if the targets where given as 1-hot encodings, then this method also returns 1-hot encodings).

`static torch_input_transforms()`

Get data augmentation pipelines for ILSVRC2012 inputs.

Note, the augmentation is inspired by the augmentation proposed in:
https://git.io/fjWPZ

Returns

Tuple containing:

- **train_transform**: A transforms pipeline that applies random transformations, normalizes the image and resizes/crops it to a final size of 224 x 224 pixels.
- **test_transform**: Similar to train_transform, but no random transformations are applied.

Return type
(tuple)

1.2.10 MNIST Dataset

The module `data.mnist_data` contains a handler for the MNIST dataset.

The implementation is based on an earlier implementation of a class I used in another project:

https://git.io/fNyQL

Information about the dataset can be retrieved from:

http://yann.lecun.com/exdb/mnist/

`class hypnettorch.data.mnist_data.MNISTData(`*data_path*, *use_one_hot=False*, *validation_size=5000*, *use_torch_augmentation=False*`)`

Bases: *Dataset*

An instance of the class shall represent the MNIST dataset.

The constructor checks whether the dataset has been read before (a pickle dump has been generated). If so, it reads the dump. Otherwise, it reads the data from scratch and creates a dump for future usage.

Note: By default, input samples are provided in a range of `[0, 1]`.

Parameters

- **data_path** (*str*) – Where should the dataset be read from? If not existing, the dataset will be downloaded into this folder.

- **use_one_hot** (*bool*) – Whether the class labels should be represented in a one-hot encoding.
- **validation_size** (*int*) – The number of validation samples. Validation samples will be taking from the training set (the first n samples).
- **use_torch_augmentation** (*bool*) – Apply data augmentation to inputs when calling method `data.dataset.Dataset.input_to_torch_tensor()`.

  The augmentation will withening the inputs according to training image statistics (mean: `0.1307`, std: `0.3081`). In training mode, it will additionally apply random crops.

  Note: If activated, the statistics of test samples are changed as a normalization is applied.

`get_identifier()`

Returns the name of the dataset.

`input_to_torch_tensor(`*x*, *device*, *mode='inference'*, *force_no_preprocessing=False*, *sample_ids=None*`)`

This method can be used to map the internal numpy arrays to PyTorch tensors.

Note, this method has been overwritten from the base class.

If enabled via constructor option `use_torch_augmentation`, input images are preprocessed. Preprocessing involves normalization and (for training mode) random perturbations.

Parameters
`(....)` – See docstring of method `data.dataset.Dataset.input_to_torch_tensor()`.

Returns
The given input `x` as PyTorch tensor.

Return type
(torch.Tensor)

`static plot_sample(`*image*, *label=None*, *interactive=False*, *file_name=None*`)`

Plot a single MNIST sample.

This method is thought to be helpful for evaluation and debugging purposes.

Deprecated since version 1.0: Please use method `data.dataset.Dataset.plot_samples()` instead.

Parameters

- **image** – A single MNIST image (given as 1D vector).
- **label** – The label of the given image.
- **interactive** – Turn on interactive mode. Thus program will run in background while figure is displayed. The figure will be displayed until another one is displayed, the user closes it or the program has terminated. If this option is deactivated, the program will freeze until the user closes the figure.
- **file_name** – (optional) If a file name is provided, then the image will be written into a file instead of plotted to the screen.

`static torch_input_transforms(`*use_random_hflips=False*`)`

Get data augmentation pipelines for MNIST inputs.

Parameters
use_random_hflips (*bool*) – Also use random horizontal flips during training.

Note: That should not be `True` for MNIST, since digits loose there meaning when flipped.

Returns

Tuple containing:

- **train_transform**: A transforms pipeline that applies random transformations and normalizes the image.
- **test_transform**: Similar to train_transform, but no random transformations are applied.

Return type
(tuple)

1.2.11 Street View House Numbers (SVHN) Dataset

The module `data.svhn_data` contains a handler for the SVHN dataset.

The dataset was introduced in:

Netzer et al., Reading Digits in Natural Images with Unsupervised Feature Learning, 2011.

This module contains a simple wrapper from the corresponding torchvision class `torchvision.datasets.SVHN` to our dataset interface `data.dataset.Dataset`.

class hypnettorch.data.svhn_data.**SVHNData**(*data_path*, *use_one_hot=False*, *validation_size=0*, *use_torch_augmentation=False*, *use_cutout=False*, *include_train_extra=False*)

Bases: *Dataset*

An instance of the class shall represent the SVHN dataset.

Note: By default, input samples are provided in a range of `[0, 1]`.

Parameters

- **data_path** (*str*) – Where should the dataset be read from? If not existing, the dataset will be downloaded into this folder.
- **use_one_hot** (*bool*) – Whether the class labels should be represented in a one-hot encoding.
- **validation_size** (*int*) – The number of validation samples. Validation samples will be taking from the training set (the first n samples).
- **use_torch_augmentation** (*bool*) – Note, this option currently only applies to input batches that are transformed using the class member *input_to_torch_tensor()* (hence, **only available for PyTorch**, so far).

  The augmentation will be identical to the one provided by class `data.cifar10_data.CIFAR10Data`, **except** that during training no random horizontal flips are applied.

Note: If activated, the statistics of test samples are changed as a normalization is applied (identical to the of class `data.cifar10_data.CIFAR10Data`).

- **use_cutout** (*bool*) – Whether option `apply_cutout` should be set of method `torch_input_transforms()`. We use cutouts of size `20 x 20` as recommended here.

  Note: Only applies if `use_data_augmentation` is set.

- **include_train_extra** (*bool*) – The training dataset can be extended by "531,131 additional, somewhat less difficult samples" (see here).

  Note, as long as the validation set size is smaller than the original training set size, all validation samples would be taken from the original training set (and thus not contain those "less difficult" samples).

get_identifier()

Returns the name of the dataset.

input_to_torch_tensor(*x*, *device*, *mode='inference'*, *force_no_preprocessing=False*, *sample_ids=None*)

This method can be used to map the internal numpy arrays to PyTorch tensors.

Note, this method has been overwritten from the base class.

The input images are preprocessed if data augmentation is enabled. Preprocessing involves normalization and (for training mode) random perturbations.

Parameters
(....) – See docstring of method `data.dataset.Dataset.input_to_torch_tensor()`.

Returns
The given input `x` as PyTorch tensor.

Return type
(torch.Tensor)

1.2.12 Udacity Self-Driving Car Challenge 2 - Steering Angle Prediction

The module *udacity_ch2* contains a handler for the Udacity Self-Driving Car Challenge 2, which contains imagery from a car's frontal center camera in combination with CAN recorded steering angles (the actual dataset contains more information, but those ingredients are enough for the steering angle prediction task).

Note: In the current implementation, this handler will not download and extract the dataset for you. You have to do this manually by following the instructions of the README file (which is located in the same folder as this file).

When using PyTorch, this class will create dataset classes (`torch.utils.data.Dataset`) for you for the training, testing and validation set. Afterwards, you can use these dataset instances to create data loaders:

```
train_loader = torch.utils.data.DataLoader(
    udacity_ch2.torch_train, batch_size=256, shuffle=True,
    num_workers=4, pin_memory=True)
```

You should then use these Pytorch data loaders rather than class internal methods to work with the dataset.

PyTorch data augmentation is applied as defined by the method `UdacityCH2Data.torch_input_transforms()`.

class hypnettorch.data.udacity_ch2.**UdacityCh2Data**(*data_path*, *num_val=0*)

Bases: *LargeImgDataset*

An instance of the class is representing the Udacity Ch2 dataset.

The input data of the dataset will be strings to image files. The output data corresponds to steering angles.

Note: The dataset has to be already downloaded and extracted before this method can be called. See the local README file for details.

Parameters

- **data_path** (*str*) – Where should the dataset be read from? The dataset folder is expected to contain the subfolders `Ch2_001` (test set) and `Ch2_002` (train and validation set). See README for details.
- **num_val** (*int*) – The number of validation samples. The validation set will be random subset of the training set. Validation samples are excluded from the training set!

  Note: Validation samples use the same data augmentation pipeline as test samples.

  ---

get_identifier()

Returns the name of the dataset.

property test_angles_available

Whether the test angles are available.

Note: If not available, test angles will all be set to zero!

The original dataset comes only with test images. However, the test set was later released too, which contains both images and angles. See the README for details.

Type
bool

tf_input_map(*mode='inference'*)

Not impemented.

static torch_input_transforms()

Get data augmentation pipelines for Udacity Ch2 inputs.

Returns

Tuple containing:

- **train_transform**: A transforms pipeline that resizes images to 256 x 192 pixels and normalizes them.
- **test_transform**: Similar to `train_transform`.

Return type
(tuple)

1.3 Sequential, custom and special datasets

1.3.1 Custom and special datasets

Contents

Continual Learning Datasets

Toy (Regression) Problems

2D Donut Dataset

This data handler creates a synthetic toy problem comprising 2D annuli.

`class` hypnettorch.data.special.donuts.**Donuts**(*centers=((0, 0), (0, 0))*, *radii=((3, 4), (9, 10))*, *num_train=100*, *num_test=100*, *use_one_hot=True*, *rseed=42*)

Bases: `Dataset`

Donut dataset handler.

Note, each donut prescribes a different class.

Parameters

- **centers** (*tuple or list*) – List of tuples, each determining the center of a donut.
- **radii** (*tuple or list*) – List of tuples, each tuple defines the inner and outer radius of a donut.
- **num_train** (*int*) – Number of training samples per donut.
- **num_test** (*int*) – Number of test samples per donut.

- **use_one_hot** (*bool*) – Whether the class labels should be represented as a one-hot encoding.
- **rseed** (*int*) – If `None`, the current random state of numpy is used to generate the data. Otherwise, a new random state with the given seed is generated.

get_identifier()

Returns the name of the dataset.

plot_dataset(*title*, *show=True*, *filename=None*, *interactive=False*, *figsize=(10, 6)*)

Plot samples belonging to this dataset.

Parameters
(....) – See docstring of method `data.dataset.Dataset.plot_samples()`.

static sample_annulus(*x_c*, *y_c*, *r_inner*, *r_outer*, *num=1*, *rand=None*)

Sample uniformly from an annulus.

Sample uniformly (x, y) satisfiying:

$$(x - x_c)^2 + (y - y_c)^2 \leq r_{\text{outer}}^2$$

and

$$(x - x_c)^2 + (y - y_c)^2 > r_{\text{inner}}^2$$

Parameters

- **x_c** (*float*) – x-position of the center.
- **y_c** (*float*) – y-position of the center.
- **r_inner** (*float*) – Inner radius.
- **r_outer** (*float*) – Outer radius.
- **num** (*int*) – Number of samples to return.
- **rand** (*numpy.random.RandomState, optional*) – Random state object used for sampling.

Returns
Array of shape `[num, 2]`.

Return type
(numpy.ndarray)

Gaussian Mixture via a set of Gaussian Datasets

The module `data.special.gaussian_mixture_data` contains a toy dataset consisting of input data drawn from a 2D Gaussian distribution. Combining several such datasets creates a Gaussian mixture (e.g., each mixture component would be one dataset from class *GaussianData*).

The dataset is inspired by the toy example provided in section 4.5 of
https://arxiv.org/pdf/1606.00704.pdf

However, the mixture of Gaussians only determines the input domain x (which is enough for a GAN dataset). Though, we also need to specify the output y.

For instance, each Gaussian bump could be the input domain of one task. Given this input domain, the task would be to predict p(x), thus y = p(x).

In the case of small variances, the task can be detected from seeing the input x alone. This allows us to predict task embeddings based on inputs, such that there is no need to define the task embedding manually.

class hypnettorch.data.special.gaussian_mixture_data.**GaussianData**(*mean=array([0, 0])*, *cov=array([[0.0025, 0.0], [0.0, 0.0025]])*, *num_train=100*, *num_test=100*, *map_function=None*, *rseed=None*)

Bases: *Dataset*

An instance of this class shall represent a regression task where the input samples x are drawn from a Gaussian with given mean and variance.

Due to plotting functionalities, this class only supports 2D inputs and 1D outputs.

Generate a new dataset.

The input data x for train and test samples will be drawn iid from the given Gaussian. Per default, the map function is the probability density of the given Gaussian: y = f(x) = p(x).

Parameters

- **mean** – The mean of the Gaussian.
- **cov** – The covariance of the Gaussian.
- **num_train** – Number of training samples.
- **num_test** – Number of test samples.
- **map_function** (*optional*) – A function handle that receives input samples and maps them to output samples. If not specified, the density function will be used as map function.
- **rseed** (*int*) – If None, the current random state of numpy is used to generate the data. Otherwise, a new random state with the given seed is generated.

property **cov**

Covariance matrix.

get_identifier()

Returns the name of the dataset.

property **mean**

Mean vector.

plot_dataset(*show=True*)

Plot the whole dataset.

Parameters
show (*bool*) – Whether the plot should be shown.

Returns
The figure handle.

static **plot_datasets**(*data_handlers*, *inputs=None*, *predictions=None*, *labels=None*, *show=True*, *filename=None*, *figsize=(10, 6)*)

Plot several datasets of this class in one plot.

Parameters

- **data_handlers** – A list of GaussianData objects.

- **inputs** (*optional*) – A list of numpy arrays representing inputs for each dataset.
- **predictions** (*optional*) – A list of numpy arrays containing the predicted output values for the given input values.
- **labels** (*optional*) – A label for each dataset.
- **show** – Whether the plot should be shown.
- **filename** (*optional*) – If provided, the figure will be stored under this filename.
- **figsize** – A tuple, determining the size of the figure in inches.

plot_predictions(*predictions*, *label='Pred'*, *show_train=True*, *show_test=True*)

Plot the dataset as well as predictions.

Parameters

- **predictions** – A tuple of x and y values, where the y values are computed by a trained regression network. Note, that x is supposed to be 2D numpy array, whereas y is a 1D numpy array.
- **label** – Label of the predicted values as shown in the legend.
- **show_train** – Show train samples.
- **show_test** – Show test samples.

plot_samples(*title*, *inputs*, *outputs=None*, *predictions=None*, *num_samples_per_row=4*, *show=True*, *filename=None*, *interactive=False*, *figsize=(10, 6)*)

Plot samples belonging to this dataset.

Note: Either `outputs` or `predictions` must be not `None`!

Parameters

- **title** – The title of the whole figure.
- **inputs** – A 2D numpy array, where each row is an input sample.
- **outputs** (*optional*) – A 2D numpy array of actual dataset targets.
- **predictions** (*optional*) – A 2D numpy array of predicted output samples (i.e., output predicted by a neural network).
- **num_samples_per_row** – Maximum number of samples plotted per row in the generated figure.
- **show** – Whether the plot should be shown.
- **filename** (*optional*) – If provided, the figure will be stored under this filename.
- **interactive** – Turn on interactive mode. We mainly use this option to ensure that the program will run in background while figure is displayed. The figure will be displayed until another one is displayed, the user closes it or the program has terminated. If this option is deactivated, the program will freeze until the user closes the figure. Note, if using the iPython inline backend, this option has no effect.
- **figsize** – A tuple, determining the size of the figure in inches.

`hypnettorch.data.special.gaussian_mixture_data.`**`get_gmm_tasks`**(*means=[array([-4, -4]), array([-4, -2]), array([-4, 0]), array([-4, 2]), array([-4, 4]), array([-2, -4]), array([-2, -2]), array([-2, 0]), array([-2, 2]), array([-2, 4]), array([0, -4]), array([0, -2]), array([0, 0]), array([0, 2]), array([0, 4]), array([2, -4]), array([2, -2]), array([2, 0]), array([2, 2]), array([2, 4]), array([4, -4]), array([4, -2]), array([4, 0]), array([4, 2]), array([4, 4])]*, *covs=[array([[0.0025, 0.0], [0.0, 0.0025]]), array([[0.0025, 0.0], [0.0, 0.0025]])]*, *num_train=100*, *num_test=100*, *map_functions=None*, *rseed=None*)

Generate a set of data handlers (one for each task) of class `GaussianData`.

Parameters

- **`means`** – The mean of each Gaussian.
- **`covs`** – The covariance matrix of each Gaussian.
- **`num_train`** – Number of training samples per task.
- **`num_test`** – Number of test samples per task.
- **`map_functions`** (*`optional`*) – A list of “map_functions”, one for each task.
- **`rseed`** (*`int`*) – See argument `rseed` of class `GaussianData`. The `i`-th dataset generated by this function will be passed the the random state `rseed+i` is specified.

Returns

A list of objects of class `GaussianData`.

Return type
(list)

Gaussian Mixture Model Dataset

The module `data.special.gaussian_mixture_data` is stemming from a conditional view, where every mode in the Gaussian mixture is a separate task (single dataset). Therefore, it provides `N` distinct data handlers when having `N` distinct modes.

Unfortunately, this configuration is not ideal for unsupervised GAN training (as we want to be able to provide batches that contain data from a mix of modes without having to manually assemble these batches) or for training a classifier for a GMM toy problem.

Therefore, this module provides a wrapper that converts a sequence of data handlers of class `data.special.gaussian_mixture_data.GaussianData` (i.e., a set of single modes) to a combined data handler.

Model description:

Let x denote the input data. The class *GMMData* assumes that it's input training data is drawn from the following Gaussian Mixture Model:

$$p(x) = \sum_{k=1}^{K} \pi_k \mathcal{N}(x; \mu_k, \Sigma_k)$$

with mixing coefficients π_k, such that $\sum_k \pi_k = 1$.

Note, it is up to the user of this class to provide appropriate training data (only important to keep in mind if unequal train set sizes are provided via constructor argument `gaussian_datasets` or if `mixing_coefficients` are non-uniform).

Let y denote a K-dimensional 1-hot encoding, i.e., $y_k \in \{0, 1\}$ and $\sum_k y_k = 1$. Thus, y is the latent variable that we want to infer (e.g., the optimal classification label) with marginal probabilities:

$$p(y_k = 1) = \pi_k$$

The conditional likelihood of a component is:

$$p(x \mid y_k = 1) = \mathcal{N}(x; \mu_k, \Sigma_k)$$

Using Bayes Theorem we obtain the posterior:

$$\begin{aligned} p(y_k = 1 \mid x) &= \frac{p(x \mid y_k = 1) p(y_k = 1)}{p(x)} \\ &= \frac{\pi_k \mathcal{N}(x; \mu_k, \Sigma_k)}{\sum_{l=1}^{K} \pi_l \mathcal{N}(x; \mu_l, \Sigma_l)} \end{aligned}$$

class hypnettorch.data.special.gmm_data.**GMMData**(*gaussian_datasets*, *classification=False*, *use_one_hot=False*, *mixing_coefficients=None*)

Bases: *Dataset*

Dataset with inputs drawn from a Gaussian mixture model.

An instance of this class combines several instances of class `data.special.gaussian_mixture_data.GaussianData` into one data handler. I.e., multiple gaussian bumps are combined to a Gaussian mixture dataset.

Most importantly, the dataset can be turned into a classification task, where the label corresponds to the ID of the Gaussian bump from which the sample was drawn. Otherwise, the original outputs will remain.

Note: You can use function `data.special.gaussian_mixture_data.get_gmm_tasks()` to create a set of tasks to be passed as constructor argument `gaussian_datasets`.

Parameters

- **gaussian_datasets** (*list*) – A list of instances of class `data.special.gaussian_mixture_data.GaussianData`.
- **classification** (*bool*) – If `True`, the original outputs of the datasets will be omitted and replaced by the dataset index. Therefore, the original regression datasets are combined to a single classification dataset.
- **use_one_hot** (*bool*) – Whether the class labels should be represented as a one-hot encoding. This option only applies if `classification` is `True`.
- **mixing_coefficients** (*list, optional*) – The mixing coefficients π_k of the individual mixture components. If not specified, π_k will be assumed to be `1. / self.num_modes`.

$$p(x) = \sum_{k=1}^{K} \pi_k \mathcal{N}(x; \mu_k, \Sigma_k)$$

Note: Mixing coefficients have to sum to 1.

Note: If mixing coefficients are not uniform, then one has to externally ensure that the training data is distributed accordingly. For instance, if `mixing_coefficients=[.1, .9]`, then the second dataset passed via `gaussian_datasets` should have 9 times more training samples then the first dataset.

estimate_distance(*fake*, *component_densities=None*, *density_estimation='hist'*, *eps=1e-05*)

This method estimates the distance/divergence of the empirical fake distribution with the underlying true data disctribution.

Therefore, we utilize the fact that we know the data distribution.

The following distance/divergence measures are implemented:

- *Symmetric KL divergence*: The fake samples are used to estimate the model density. The fake samples are used to estimate $D_{\text{KL}}(\text{fake} \mid\mid \text{real})$. An additional set of real samples is drawn from the training data to compute a Monte Carlo estimate of $D_{\text{KL}}(\text{real} \mid\mid \text{fake})$.

  Comment from Simone Surace about this approach: "Doing density estimation first and then computing the integral is known to be the wrong way to go (there is an entire literature about this problem)." This should be kept in mind when using this estimate.

  Parameters

  - **fake** (*numpy.ndarray*) – A 2D numpy array, where each row is an input sample (usually drawn from a generator network).
  - **component_densities** (*numpy.ndarray, optional*) – A 2D numpy array with each row corresponding to a sample in `fake` and each column corresponding to a mode in this dataset. Each entry represents the density of the corresponding sample under the corresponding mixture component. See return value `responsibilities` of method *estimate_mode_coverage()*.

- **density_estimation** – Which kind of method should be used to estimate the model distribution (i.e., density of given samples under the distribution estimated from those samples). Available methods are:
  - `'hist'`: We estimate the fake density based on a normalized 2D histogram of the samples. We use the Square-root choice to compute the number of bins per dimension.
  - `'gaussian'`: Uses the kernel density method `'gaussian'` from `sklearn.neighbors.kde.KernelDensity`. Note, we don't change the default `bandwidth` value!
- **eps** (*float*) – We don't allow densities to be smaller than this value for numerical stability reasons (when computing the log).

Returns
The estimated symmetric KL divergence.

`estimate_mode_coverage`(*fake*, *responsibilities=None*)

Compute the mode coverage of fake samples as suggested in

https://arxiv.org/abs/1606.00704

This method will compute the responsibilities for each fake sample towards each mixture component and assign each sample to the mixture component with the highest responsibility. Mixture components that get no fake sample assigned are considered dropped modes.

The paper referenced above used 10,000 fake samples (on their synthetic dataset) to measure the mode coverage.

Parameters

- **fake** – A 2D numpy array, where each row is an input sample (usually drawn from a generator network).
- **responsibilities** (*optional*) – The responsibilities of each *fake* data point (may be unnormalized). A 2D numpy array with each row corresponding to a sample in *fake* and each column corresponding to a mode in this dataset.

Returns

A tuple containing:

- **num_covered**: The number of modes that have at least one fake sample with maximum responsibility being assigned to that mode.
- **responsibilities**: The given or computed *responsibilities*. If computed by this method, the responsibilities will be unnormalized, i.e., correspond to the densities per component of this mixture model.

Return type
(tuple)

`get_identifier`()

Returns the name of the dataset.

`get_input_mesh`(*x1_range=None*, *x2_range=None*, *grid_size=1000*)

Create a 2D grid of input values.

The default grid returned by this method will also be the default grid used by the method *plot_uncertainty_map()*.

Note: This method is only implemented for 2D datasets.

Parameters

- **x1_range** (*tuple, optional*) – The min and max value for the first input dimension. If not specified, the range will be automatically inferred.

  Automatical inference is based on the underlying data (train and test). The range will be set, such that all data can be drawn inside.

- **x2_range** (*tuple, optional*) – Same as `x1_range` for the second input dimension.
- **grid_size** (*int or tuple*) – How many input samples per dimension. If an integer is passed, then the same number grid size will be used for both dimension. The grid is build by equally spacing `grid_size` inside the ranges `x1_range` and `x2_range`.

Returns

Tuple containing:

- **x1_grid** (numpy.ndarray): A 2D array, containing the grid values of the first dimension.
- **x2_grid** (numpy.ndarray): A 2D array, containing the grid values of the second dimension.
- **flattended_grid** (numpy.ndarray): A 2D array, containing all samples from the first dimension in the first column and all values corresponding to the second dimension in the second column. This format correspond to the input format as, for instance, returned by methods such as `data.dataset.Dataset.get_train_inputs()`.

Return type
(tuple)

property means

2D array, containing the mean of each component in its rows.

Type
np.ndarray

property num_modes

The number of mixture components.

Type
int

plot_optimal_classification(*title='Classification Map'*, *input_mesh=None*, *mesh_modes=None*, *sample_inputs=None*, *sample_modes=None*, *sample_label=None*, *sketch_components=False*, *show=True*, *filename=None*, *figsize=(10, 6)*)

Plot a color-coded grid on how to optimally classify for each input value.

Note: Since the training data is drawn randomly, it might be that some training samples have a label that doesn't correpond to the optimal label.

Parameters

- **(....)** – See arguments of method *plot_uncertainty_map()*.

- **mesh_modes** (*numpy.ndarray, optional*) – If not provided, then the color of each grid position x is determined based on $\arg\max_k \pi_k \mathcal{N}(x; \mu_k, \Sigma_k)$. Otherwise, the labeling provided here will determine the coloring.

plot_real_fake(*title*, *real*, *fake*, *show=True*, *filename=None*, *interactive=False*, *figsize=(10, 6)*)

Useful method when using this dataset in conjunction with GAN training. Plots the given real and fake input samples in a 2D plane.

Parameters

- **(....)** – See docstring of method `data.dataset.Dataset.plot_samples()`.
- **real** (*numpy.ndarray*) – A 2D numpy array, where each row is an input sample. These samples correspond to actual input samples drawn from the dataset.
- **fake** (*numpy.ndarray*) – A 2D numpy array, where each row is an input sample. These samples correspond to generated samples.

plot_samples(*title*, *inputs*, *outputs=None*, *predictions=None*, *show=True*, *filename=None*, *interactive=False*, *figsize=(10, 6)*)

Plot samples belonging to this dataset.

Parameters

(....) – See docstring of method `data.dataset.Dataset.plot_samples()`.

plot_uncertainty_map(*title='Uncertainty Map'*, *input_mesh=None*, *uncertainties=None*, *use_generative_uncertainty=False*, *use_ent_joint_uncertainty=False*, *sample_inputs=None*, *sample_modes=None*, *sample_label=None*, *sketch_components=False*, *norm_eps=None*, *show=True*, *filename=None*, *figsize=(10, 6)*)

Draw an uncertainty heatmap.

Parameters

- **title** (*str*) – Title of plots.
- **input_mesh** (*tuple, optional*) – The input mesh of the heatmap (see return value of method *get_input_mesh()*). If not specified, the default return value of method *get_input_mesh()* is used.
- **uncertainties** (*numpy.ndarray, optional*) – The uncertainties corresponding to `input_mesh`. If not specified, then the uncertainties will be computed based the entropy across $k = 1..K$ for

$$p(y_k = 1 \mid x) = \frac{\pi_k \mathcal{N}(x; \mu_k, \Sigma_k)}{\sum_{l=1}^{K} \pi_l \mathcal{N}(x; \mu_l, \Sigma_l)}$$

Note: The entropies will be normalized by the maximum uncertainty `-np.log(1.0 / self.num_modes)`.

- **use_generative_uncertainty** (*bool*) – If `True`, the uncertainties plotted by default (if `uncertainties` is left unspecified) are not based on the entropy of the responsibilities $p(y_k = 1 \mid x)$, but are the densities of the underlying GMM $p(x)$.
- **use_ent_joint_uncertainty** (*bool*) – If `True`, the uncertainties plotted by default (if

uncertainties is left unspecified) are based on the entropy of $p(y, x)$ at location x:

$$
\begin{aligned}
& -\sum_k p(x)p(y_k = 1 \mid x) \log p(x)p(y_k = 1 \mid x) \\
= & -p(x) \sum_k p(y_k = 1 \mid x) \log p(y_k = 1 \mid x) - p(x) \log p(x)
\end{aligned}
$$

Note, we normalize $p(x)$ by its maximum inside the chosen grid. Hence, the plot depends on the chosen input_mesh. In this way, $p(x) \in [0, 1]$ and the second term $-p(x) \log p(x) \in [0, \exp(-1)]$ (note, $-p(x) \log p(x)$ would be negative for $p(x) > 1$).

The first term is simply the entropy of $p(y \mid x)$ scaled by $p(x)$. Hence, it shows where in the input space are the regions where Gaussian bumps are overlapping (regions in which data exists but multiple labels y are possible).

The second term shows the boundaries of the data manifold. Note, $-1 \log 1 = 0$ and $-\lim_{p(x) \to 0} p(x) \log p(x) = 0$.

Note: This option is mutually exclusive with option use_generative_uncertainty.

Note: Entropies of $p(y \mid x)$ won't be normalized in this case.

- **sample_inputs** (*numpy.ndarray, optional*) – Sample inputs. Can be specified if a scatter plot of samples (e.g., train samples) should be laid above the heatmap.
- **sample_modes** (*numpy.ndarray, optional*) – To which mode do the samples in sample_inputs belong to? If provided, then for each sample in sample_inputs a number smaller than *num_modes* is expected. All samples with the same mode identifier are colored with the same color.
- **sample_label** (*str, optional*) – If a label should be shown in the legend for inputs sample_inputs.
- **sketch_components** (*bool*) – Sketch the mean and variance of each component.
- **norm_eps** (*float, optional*) – If uncertainties are computed by this method, then (normalized) densities for each x-value in the input mesh have to be computed. To avoid division by zero, a positive number norm_eps can be specified.
- **(....)** – See docstring of method data.dataset.Dataset.plot_samples().

1D Regression Dataset

The module data.special.regression1d_data contains a data handler for a CL toy regression problem. The user can construct individual datasets with this data handler and use each of these datasets to train a model in a continual leraning setting.

class hypnettorch.data.special.regression1d_data.**ToyRegression**(*train_inter=[-10, 10], num_train=20, test_inter=[-10, 10], num_test=80, val_inter=None, num_val=None, map_function=<function ToyRegression.<lambda>>, std=0.0, perturb_test_val=False, rseed=None*)

Bases: *Dataset*

An instance of this class shall represent a simple regression task.

Generate a new dataset.

The input data x will be uniformly drawn for train samples and equidistant for test samples. The user has to specify a function that will map this random input data onto output samples y.

Parameters

- **train_inter** (*tuple or list*) – A tuple, representing the interval from which x samples are drawn in the training set.

  `train_inter` may also be provided as a list of tuples, in which case training samples will be distributed according to the range covered by each tuple.
- **num_train** (*int*) – Number of training samples.
- **test_inter** (*tuple*) – A tuple, representing the interval from which x samples are drawn in the test set.
- **num_test** (*int*) – Number of test samples.
- **val_inter** (*tuple, optional*) – See parameter *test_inter*. If set, this argument leads to the construction of a validation set. Note, option `num_val` need to be specified as well.
- **num_val** (*int, optional*) – Number of validation samples.
- **map_function** (*func*) – A function handle that receives input samples and maps them to output samples.
- **std** (*float or func*) – If not zero, Gaussian white noise with this std will be added to the training outputs.

  Heteroscedasticity can be realized by passing a function $\sigma(x)$ that describes the standard deviations at a given location x. Note, this function may only outputs numbers ≥ 0.
- **perturb_test_val** (*bool*) – By default, the option `std` only adds noise to the training data, not the validation or test data. If this option is `True`, then also the validation and test targets will be perturbed. This might be helpful for measuring calibration.
- **rseed** (*int*) – If `None`, the current random state of numpy is used to generate the data. Otherwise, a new random state with the given seed is generated.

get_identifier()

Returns the name of the dataset.

plot_dataset(*show=True*)

Plot the whole dataset.

Parameters

show – Whether the plot should be shown.

static plot_datasets(*data_handlers*, *inputs=None*, *predictions=None*, *labels=None*, *fun_xranges=None*, *show=True*, *filename=None*, *figsize=(10, 6)*, *publication_style=False*)

Plot several datasets of this class in one plot.

Parameters

- **data_handlers** – A list of ToyRegression objects.
- **inputs** (*optional*) – A list of numpy arrays representing inputs for each dataset.

- **predictions** (*optional*) – A list of numpy arrays containing the predicted output values for the given input values.
- **labels** (*optional*) – A label for each dataset.
- **fun_xranges** (*optional*) – List of x ranges in which the true underlying function per dataset should be sketched.
- **show** – Whether the plot should be shown.
- **filename** (*optional*) – If provided, the figure will be stored under this filename.
- **figsize** – A tuple, determining the size of the figure in inches.
- **publication_style** – Whether the plots should be in publication style.

plot_predictions(*predictions*, *label='Pred'*, *show_train=True*, *show_test=True*)

Plot the dataset as well as predictions.

Parameters

- **predictions** – A tuple of x and y values, where the y values are computed by a trained regression network. Note, that we assume the x values to be sorted.
- **label** – Label of the predicted values as shown in the legend.
- **show_train** – Show train samples.
- **show_test** – Show test samples.

plot_samples(*title*, *inputs*, *outputs=None*, *predictions=None*, *num_samples_per_row=4*, *show=True*, *filename=None*, *interactive=False*, *figsize=(10, 6)*)

Plot samples belonging to this dataset.

Note: Either `outputs` or `predictions` must be not `None`!

Parameters

- **title** – The title of the whole figure.
- **inputs** – A 2D numpy array, where each row is an input sample.
- **outputs** (*optional*) – A 2D numpy array of actual dataset targets.
- **predictions** (*optional*) – A 2D numpy array of predicted output samples (i.e., output predicted by a neural network).
- **num_samples_per_row** – Maximum number of samples plotted per row in the generated figure.
- **show** – Whether the plot should be shown.
- **filename** (*optional*) – If provided, the figure will be stored under this filename.
- **interactive** – Turn on interactive mode. We mainly use this option to ensure that the program will run in background while figure is displayed. The figure will be displayed until another one is displayed, the user closes it or the program has terminated. If this option is deactivated, the program will freeze until the user closes the figure. Note, if using the iPython inline backend, this option has no effect.
- **figsize** – A tuple, determining the size of the figure in inches.

property test_x_range

The input range for test samples.

property train_x_range

The input range for training samples.

property val_x_range

The input range for validation samples.

1D Regression Dataset with bimodal error

The module `data.special.regression1d_bimodal_data` contains a data handler for a CL toy regression problem. The user can construct individual datasets with this data handler and use each of these datasets to train a model in a continual learning setting.

class hypnettorch.data.special.regression1d_bimodal_data.**BimodalToyRegression**(*train_inter=[-10, 10], num_train=20, test_inter=[-10, 10], num_test=80, val_inter=None, num_val=None, map_function=<function BimodalToyRegression.<lambda>>, alpha1=0.5, dist1=5, dist2=None, std1=1, std2=None, rseed=None, perturb_test_val=False*)

Bases: *ToyRegression*

An instance of this class shall represent a simple regression task, but with a bimodal Gaussian mixture error distribution.

Generate a new dataset.

The input data x will be uniformly drawn for train samples and equidistant for test samples. The user has to specify a function that will map this random input data onto output samples y.

Parameters

- **(....)** – See docstring of class `data.special.regression_1d_data.ToyRegression`.
- **alpha1** – Mixture coefficient of the first Gaussian mode of the error.
- **dist1** – The distance from zero of mean of the first Gaussian component of the error.
- **dist2** (*optional*) – The distance from zero of mean of the first Gaussian component of the error. If `None`, the value of *dist1* will be taken.
- **std1** – The standard deviation of the first Gaussian component of the error.

- **std2** (*optional*) – The standard deviation of the first Gaussian component of the error. If None, the value of *std1* will be taken.

`get_identifier()`

Returns the name of the dataset.

Classification Tasks

Permuted MNIST Dataset

The module `data.special.permuted_mnist` contains a data handler for the permuted MNIST dataset.

`class hypnettorch.data.special.permuted_mnist.PermutedMNIST(`*data_path*, *use_one_hot=True*, *validation_size=0*, *permutation=None*, *padding=0*, *trgt_padding=None*`)`

Bases: *MNISTData*

An instance of this class shall represent the permuted MNIST dataset, which is the same as the MNIST dataset, just that input pixels are shuffled by a random matrix.

Note: Image transformations are computed on the fly when transforming batches to torch tensors. Hence, this class is only applicable to PyTorch applications. Internally, the class stores the unpermuted images.

Parameters

- **data_path** – Where should the dataset be read from? If not existing, the dataset will be downloaded into this folder.
- **use_one_hot** – Whether the class labels should be represented in a one-hot encoding.
- **validation_size** – The number of validation samples. Validation samples will be taking from the training set (the first n samples).
- **permutation** – The permutation that should be applied to the dataset. If None, no permutation will be applied. We expect a numpy permutation of the form `np.random.permutation((28+2*padding)**2)`
- **padding** – The amount of padding that should be applied to images.

  Note: The padding is currently not reflected in the *:attr:`data.dataset.Dataset.in_shape* attribute, as the padding is only applied to torch tensors. See attribute *torch_in_shape*.

- **trgt_padding** (*int, optional*) – If provided, `trgt_padding` fake classes will be added, such that in total the returned dataset has `len(labels) + trgt_padding` classes. However, all padded classes have no input instances. Note, that 1-hot encodings are padded to fit the new number of classes.

`get_identifier()`

Returns the name of the dataset.

input_to_torch_tensor(*x*, *device*, *mode='inference'*, *force_no_preprocessing=False*, *sample_ids=None*)

This method can be used to map the internal numpy arrays to PyTorch tensors.

Note, this method has been overwritten from the base class.

It applies zero padding and pixel permutations.

Parameters

(....) – See docstring of method `data.dataset.Dataset.input_to_torch_tensor()`.

Returns

The given input `x` as PyTorch tensor.

Return type

(torch.Tensor)

property permutation

The permuation matrix that is applied to input images before they are transformed to Torch tensors.

tf_input_map(*mode='inference'*)

Not implemented! The class currently does not support Tensorflow.

property torch_in_shape

The input shape of images, similar to attribute *in_shape*. In contrast to *in_shape*, this attribute reflects the padding that is applied when calling `classifier.permuted_mnist.PermutedMNIST.input_to_torch_tensor()`.

static torch_input_transforms(*permutation=None*, *padding=0*)

Transform MNIST images to PyTorch tensors.

Parameters

- **permutation** – A given permutation that should be applied to all images.
- **padding** – Apply a given amount of zero padding.

Returns

A transforms pipeline.

class hypnettorch.data.special.permuted_mnist.**PermutedMNISTList**(*permutations*, *data_path*, *use_one_hot=True*, *validation_size=0*, *padding=0*, *trgt_padding=None*, *show_perm_change_msg=True*)

Bases: object

A list of permuted MNIST tasks that only uses a single instance of class *PermutedMNIST*.

An instance of this class emulates a Python list that holds objects of class *PermutedMNIST*. However, it doesn't actually hold several objects, but only one with just the permutation matrix being exchanged everytime a different element of this list is retrieved. Therefore, **use this class with care**!

- As all list entries are the same PermutedMNIST object, one should never work with several list entries at the same time! -> **Retrieving a new list entry will modify every previously retrieved list entry!**
- When retrieving a slice, a shallow copy of this object is created (i.e., the underlying *PermutedMNIST* does not change) with only the desired subgroup of permutations avaliable.

Why would one use this object? When working with many permuted MNIST tasks, then the memory consumption becomes significant if one desires to hold all task instances at once in working memory. An object of this

class only needs to hold the MNIST dataset once in memory. Just the number of permutation matrices grows linearly with the number of tasks.

Caution: You may never use more than one entry of this class at the same time, as all entries share the same underlying data object and therewith the same permutation.

Note: The mini-batch generation process is maintained separately for every permutation. Thus, the retrieval of mini-batches for different permutations does not influence one another.

Example

You should **never** use this list as follows

```
dhandlers = PermutedMNISTList(permutations, '/tmp')
d0 = dhandlers[0]
# Zero-th permutation is active ...
# ...
d1 = dhandlers[1]
# First permutation is active for `d0` and `d1`!
# Important, you may not use `d0` anymore, as this might lead to
# undesired behavior.
```

Example

Instead, always work with only one list entry at a time. The following usage would be **correct**

```
dhandlers = PermutedMNISTList(permutations, '/tmp')
d = dhandlers[0]
# Zero-th permutation is active ...
# ...
d = dhandlers[1]
# First permutation is active for `d` as expected.
```

Parameters

- **(....)** – See docstring of constructor of class *PermutedMNIST*.
- **permutations** – A list of permutations (see parameter `permutation` of class *PermutedMNIST* to have a description of valid list entries). The length of this list denotes the number of tasks.
- **show_perm_change_msg** – Whether to print a notification everytime the data permutation has been exchanged. This should be enabled during developement such that a proper use of this list is ensured. **Note** You may never work with two elements of this list at a time.

Split MNIST Dataset

The module `data.special.split_mnist` contains a wrapper for data handlers for the SplitMNIST task.

class hypnettorch.data.special.split_mnist.**SplitMNIST**(*data_path*, *use_one_hot=False*, *validation_size=1000*, *use_torch_augmentation=False*, *labels=[0, 1]*, *full_out_dim=False*, *trgt_padding=None*)

Bases: *MNISTData*

An instance of the class shall represent a SplitMNIST task.

Parameters

- **data_path** (*str*) – Where should the dataset be read from? If not existing, the dataset will be downloaded into this folder.
- **use_one_hot** (*bool*) – Whether the class labels should be represented in a one-hot encoding.
- **validation_size** (*int*) – The number of validation samples. Validation samples will be taking from the training set (the first n samples).
- **use_torch_augmentation** (*bool*) – See docstring of class `data.mnist_data.MNISTData`.
- **labels** (*list*) – The labels that should be part of this task.
- **full_out_dim** (*bool*) – Choose the original MNIST instead of the new task output dimension. This option will affect the attributes `data.dataset.Dataset.num_classes` and `data.dataset.Dataset.out_shape`.
- **trgt_padding** (*int, optional*) – If provided, `trgt_padding` fake classes will be added, such that in total the returned dataset has `len(labels) + trgt_padding` classes. However, all padded classes have no input instances. Note, that 1-hot encodings are padded to fit the new number of classes.

get_identifier()

Returns the name of the dataset.

transform_outputs(*outputs*)

Transform the outputs from the 10D MNIST dataset into proper labels based on the constructor argument `labels`.

I.e., the output will have `len(labels)` classes.

Example

Split with labels [2,3]

1-hot encodings: [0,0,0,1,0,0,0,0,0,0] -> [0,1]

labels: 3 -> 1

Parameters
outputs – 2D numpy array of outputs.

Returns
2D numpy array of transformed outputs.

hypnettorch.data.special.split_mnist.**get_split_mnist_handlers**(*data_path*, *use_one_hot=True*, *validation_size=0*, *use_torch_augmentation=False*, *num_classes_per_task=2*, *num_tasks=None*, *trgt_padding=None*)

This function instantiates 5 objects of the class *SplitMNIST* which will contain a disjoint set of labels.

The SplitMNIST task consists of 5 tasks corresponding to the images with labels [0,1], [2,3], [4,5], [6,7], [8,9].

Parameters

- **data_path** – Where should the MNIST dataset be read from? If not existing, the dataset will be downloaded into this folder.
- **use_one_hot** – Whether the class labels should be represented in a one-hot encoding.
- **validation_size** – The size of the validation set of each individual data handler.
- **use_torch_augmentation** (*bool*) – See docstring of class `data.mnist_data.MNISTData`.
- **num_classes_per_task** (*int*) – Number of classes to put into one data handler. If 2, then every data handler will include 2 digits.
- **num_tasks** (*int, optional*) – The number of data handlers that should be returned by this function.
- **trgt_padding** (*int, optional*) – See docstring of class *SplitMNIST*.

Returns
A list of data handlers, each corresponding to a *SplitMNIST* object.

Return type
(list)

Split CIFAR-10/100 Dataset

The module `data.special.split_cifar` contains a wrapper for data handlers for the Split-CIFAR10/CIFAR100 task.

class hypnettorch.data.special.split_cifar.**SplitCIFAR100Data**(*data_path*, *use_one_hot=False*, *validation_size=1000*, *use_data_augmentation=False*, *use_cutout=False*, *labels=range(0, 10)*, *full_out_dim=False*)

Bases: *CIFAR100Data*

An instance of the class shall represent a single SplitCIFAR-100 task.

Parameters

- **data_path** – Where should the dataset be read from? If not existing, the dataset will be downloaded into this folder.
- **use_one_hot** (*bool*) – Whether the class labels should be represented in a one-hot encoding.
- **validation_size** – The number of validation samples. Validation samples will be taking from the training set (the first n samples).

- **use_data_augmentation** (*optional*) – Note, this option currently only applies to input batches that are transformed using the class member `data.dataset.Dataset.input_to_torch_tensor()` (hence, **only available for PyTorch**). Note, we are using the same data augmentation pipeline as for CIFAR-10.
- **use_cutout** (*bool*) – See docstring of class `data.cifar10_data.CIFAR10Data`.
- **labels** – The labels that should be part of this task.
- **full_out_dim** – Choose the original CIFAR instead of the the new task output dimension. This option will affect the attributes `data.dataset.Dataset.num_classes` and `data.dataset.Dataset.out_shape`.

get_identifier()

Returns the name of the dataset.

transform_outputs(*outputs*)

Transform the outputs from the 100D CIFAR100 dataset into proper labels based on the constructor argument `labels`.

See `data.special.split_mnist.SplitMNIST.transform_outputs()` for more information.

Parameters
outputs – 2D numpy array of outputs.

Returns
2D numpy array of transformed outputs.

class hypnettorch.data.special.split_cifar.**SplitCIFAR10Data**(*data_path*, *use_one_hot=False*, *validation_size=1000*, *use_data_augmentation=False*, *use_cutout=False*, *labels=range(0, 2)*, *full_out_dim=False*)

Bases: *CIFAR10Data*

An instance of the class shall represent a single SplitCIFAR-10 task.

Each instance will contain only samples of CIFAR-10 belonging to a subset of the labels.

Parameters
(....) – See docstring of class *SplitCIFAR100Data*.

get_identifier()

Returns the name of the dataset.

transform_outputs(*outputs*)

Transform the outputs from the 10D CIFAR10 dataset into proper labels based on the constructor argument `labels`.

See `data.special.split_mnist.SplitMNIST.transform_outputs()` for more information.

Parameters
outputs (*numpy.ndarray*) – 2D numpy array of outputs.

Returns
2D numpy array of transformed outputs.

Return type
(numpy.ndarray)

hypnettorch.data.special.split_cifar.**get_split_cifar_handlers**(*data_path*, *use_one_hot=True*, *validation_size=0*, *use_data_augmentation=False*, *use_cutout=False*, *num_classes_per_task=10*, *num_tasks=6*)

This method will combine 1 object of the class `data.cifar10_data.CIFAR10Data` and 5 objects of the class *SplitCIFAR100Data*.

The SplitCIFAR benchmark consists of 6 tasks, corresponding to the images in CIFAR-10 and 5 tasks from CIFAR-100 corresponding to the images with labels [0-10], [10-20], [20-30], [30-40], [40-50].

Parameters

- **data_path** – Where should the CIFAR-10 and CIFAR-100 datasets be read from? If not existing, the datasets will be downloaded into this folder.
- **use_one_hot** (*bool*) – Whether the class labels should be represented in a one-hot encoding.
- **validation_size** – The size of the validation set of each individual data handler.
- **use_data_augmentation** (*optional*) – Note, this option currently only applies to input batches that are transformed using the class member `data.dataset.Dataset.input_to_torch_tensor()` (hence, **only available for PyTorch**).
- **use_cutout** (*bool*) – See docstring of class `data.cifar10_data.CIFAR10Data`.
- **num_classes_per_task** (*int*) – Number of classes to put into one data handler. For example, if `2`, then every data handler will include 2 digits.

  If `10`, then the first dataset will simply be CIFAR-10.
- **num_tasks** (*int*) – A number between 1 and 11 (assuming `num_classes_per_task == 10`), specifying the number of data handlers to be returned. If `num_tasks=6`, then there will be the CIFAR-10 data handler and the first 5 splits of the CIFAR-100 dataset (as in the usual CIFAR benchmark for CL).

Returns

(list) A list of data handlers. The first being an instance of class `data.cifar10_data.CIFAR10Data` and the remaining ones being an instance of class *SplitCIFAR100Data*.

1.3.2 Timeseries Datasets

Contents

Common Datasets

Dataset for the sequential copy task

A data handler for the copy task as described in:

https://arxiv.org/pdf/1410.5401.pdf

A typical usecase of this dataset is in an incremental learning setting. For instance, a sequence of tasks with increasing lengths can be used in curriculum learning or continual learning.

The class contains a lot of options to modify the basic copy task. Many of those variations target the usecase continual learning (rather than curriculum learning) by providing sets of distinct tasks with comparable difficulty. Note, these variations typically extend the required input processing and are not limited to plain copying.

class hypnettorch.data.timeseries.copy_data.**CopyTask**(*min_input_len*, *max_input_len*, *seq_width=7*, *out_width=-1*, *num_train=100*, *num_test=100*, *num_val=None*, *pat_len=-1*, *scatter_pattern=False*, *permute_width=False*, *permute_time=False*, *permute_xor=False*, *permute_xor_iter=1*, *permute_xor_separate=False*, *random_pad=False*, *pad_after_stop=False*, *pairwise_permute=False*, *revert_output_seq=False*, *rseed=None*, *rseed_permute=None*, *rseed_scatter=None*)

Bases: *SequentialDataset*

Data handler for the sequential copy task.

In this task, a binary vector is presented as input, and the network has to learn to copy it. Such that the network cannot rely on intermediate information, there is a delay between the end of the input presentation and the output generation. The end of the input sequence is delimited by a binary bit, which is always zero except when the sequence finishes. This flag should not be copied.

An instance of this class will represent copy task patterns of random length (by default) but fixed width (but see option `out_width`). The length of input patterns will be sampled uniformly from the interval `[min_input_len, max_input_len]`. Note that the actual length of the patterns `pat_len` might be smaller in the case where there are a certain number of zero-valued timesteps within the input. As such, every sequence is characterised by the following values:

- `pat_len`: the actual length of the binary pattern to be copied. Across this duration, half the pixels have value of 1 and the other half have value 0.
- `input_len`: the length of input presentation up until the stop flag. It is equal to the pattern length plus the number of zero-valued timesteps.

- `seq_len`: the length of the entire sequences, including input presentation, stop flag and output generation. Therefore it is equal to the input length, plus one (stop flag), plus the pattern length (since during output reconstruction we don't care about reconstructing the zero-valued part of the input).

Caution: Manipulations such as permutations or scattering/masking will be applied online in *output_to_torch_tensor()*.

Parameters

- **min_input_len** (*int*) – The minimum length of an input sequence.

  Note: The input length is the length of the presented input before the stop flag. It might include both a pattern to be copied and a set of zero-valued timesteps that do not need to be reconstructed.

- **max_input_len** (*int*) – The maximum length of a pattern.
- **seq_width** (*int*) – The width if each pattern.

  Note: Each pattern will have a certain length (across time) and a certain width.

- **out_width** (*int, optional*) – If specified, a number smaller than `seq_width` is expected. In this case, only the first `out_width` input features are expected to be copied (i.e., only those occur as target output features).
- **num_train** (*int*) – Number of training samples.
- **num_test** (*int*) – Number of test samples.
- **num_val** (*int, optional*) – Number of validation samples.
- **pat_len** (*int, optional*) – The actual length of the pattern within the input sequence (excluding zero-valued timesteps). By default, the value is `-1` meaning that the pattern length is identical to the input length, and there are no zeroed timesteps. For other values, the input sequences will be zero-padded after `pat_len` timesteps. Therefore, the input sequence lengths remain the same, but the actual duration of the patterns is reduced. This manipulation is useful to decouple sequence length and memory requirement for analysis.

  Note: We define the number of timesteps that are not zero, and therefore for values different than `-1` with the current implementation we will obtain patterns of identical length (but different input sequence length).

- **scatter_pattern** (*bool*) – Option only compatible with `pat_len != -1`. If activated, the pattern is not concentrated at the beginning of the input sequence. Instead, the whole input sequence will be filled with a random pattern (i.e., no padding is used) but only a fixed and random (see option `rseed_scatter`) number of timesteps from the input sequence are considered to create an output sequence of length `pat_len`.
- **permute_width** (*boolean, optional*) – If enabled, the generated pattern will be permuted along the width axis.
- **permute_time** (*boolean, optional*) – If enabled, the generated pattern will be permuted along the temporal axis.

- **permute_xor** (*bool*) – Only applicable if `permute_width` or `permute_time` is `True`. If `True`, the permuted and unpermuted output pattern will be combined to a new output pattern via a logical xor operation.
- **permute_xor_iter** (*int*) – Only applicable if `permute_xor` is set. If `True`, the internal permutation is applied iteratively and XOR-ed with the previous target output to obtain a final target output.
- **permute_xor_separate** (*bool*) – Only applicable if `permute_xor` is set and `permute_xor_iter > 1`. If `True`, a separate permutation matrix is used per iteration described by `permute_xor_iter`. In this case, we the input pattern is `permute_xor_iter` times permuted via a separate permutation matrix and the resulting patterns are sequentially XOR-ed with the original input pattern.

  Hence, this can be viewed as follows: `permute_xor_iter` random input pixels are assigned to each output pattern pixel. This output pattern pixel will be 1 if and only if the number of ones in those input pixels is odd.
- **random_pad** (*bool, optional*) – If activated, the truncated part of the input (see option `pat_len`) will be left as a random pattern, and not set to zero. Note that the loss computation is unaffected by this option.
- **pad_after_stop** (*bool*) – This option will affect how option `pat_len` is handled and therefore can only be used if `pat_len` is set. If `True`, `pat_len` will determine the length of the input sequence (no padding applied before the stop bit). Therefore, the padding is moved to after the stop bit and therewith part of the target output. I.e., the original input sequence length determines the output sequence length which consists of zero padding and the input pattern of length `pat_len`. Note, in this case, the options `min_input_len` and `max_input_len` actually apply solely to the output.
- **pairwise_permute** (*bool, optional*) – This option is only used if some permutation is activated. If enabled, it will force the permutation to be a pairwise switch between successive pixels. Note that this operation is deterministic, and will therefore be identical for different tasks, if more than one task is generated.
- **revert_output_seq** (*bool, optional*) – If enabled, it will revert output sequences along the time dimension. Note that this operation is deterministic, and will therefore be identical for different tasks, if more than one task is generated.
- **rseed** (*int, optional*) – If `None`, the current random state of numpy is used to generate the data. Otherwise, a new random state with the given seed is generated.
- **rseed_permute** (*int, optional*) – Random seed for performing permutations of the copy patterns. Only used if option `permute_width` or `permute_time` are activated. If `None`, the current random state of numpy is used to generate the data. Otherwise, a new random state with the given seed is generated.
- **rseed_scatter** (*int, optional*) – See option `rseed`. Random seed for determining which timesteps of the input sequence to use for the output pattern if option `scatter_pattern` is activated.

static create_permutation_matrix(*permute_time*, *permute_width*, *pat_len_perm*, *seq_width*, *rstate_permute*, *pairwise_permute=False*, *revert_output_seq=False*)

Create a permutation matrix.

Parameters

- **pairwise_permute** (*boolean, optional*) – If True, the permutations correspond to switching the position of neighboring pixels. For example *1234567* would become *2143657*. If the number of timesteps is odd, the last timestep is left unmoved.
- **revert_output_seq** (*boolean, optional*) – If True, the output sequences will be inverted along the time dimension. I.e. a pattern *1234567* would become *7654321*.

get_identifier()

Returns the name of the dataset.

get_out_pattern_bounds(*sample_ids*)

Get the start time step and length of the output pattern within the sequence.

Note, input sequences may have varying length (even though they are padded to the same length). Assume we are considering a input of length 7, meaning that the total sequence would have the length 15 = 7 + 1 + 7 (input pattern presentation, stop bit, output pattern copying). In addition, assume that the maximum input length is 10 (hence, the maximum input length is 21 = 10 + 1 + 10). In this case, all sequences are padded to have length 21. For the sample in consideration (with input length 7), the output pattern sequence starts at index 8 and has a length of 7, or less, if the input contains some zeroed values. Hence, these two number would be returned for this sample.

Parameters
(....) – See docstring of method `data.sequential_data.SequentialDataset.get_in_seq_lengths()`.

Returns

Tuple containing:

- **start_inds** (numpy.ndarray): 1D array with the same length as `sample_ids`, which contains the start index for output pattern in a given sample.
- **lengths** (numpy.ndarray): 1D array containing the lengths of the pattern per given sample.

Return type
(tuple)

get_zeroed_ts(*sample_ids*)

Get the number of zeroed timesteps in each input pattern.

Note, if `scatter_pattern` was activated in the constructor, then this number does not refer to the number of padded steps in the input sequence but rather to the number of unused steps in the input sequence. However, those unused steps will still contain random patterns. Similarly, if argument `random_pad` is used.

Note, if `pad_after_stop` was activated, then the zeroed timesteps actually occur after the stop bit, i.e., in the output part of the sequence.

Parameters
(....) – See docstring of method `get_in_seq_lengths()`.

Returns
A 1D numpy array.

Return type
(numpy.ndarray)

output_to_torch_tensor(**args*, ***kwargs*)

Similar to method `input_to_torch_tensor()`, just for dataset outputs.

Parameters
(....) – See docstring of method `data.dataset.Dataset.output_to_torch_tensor()`.

Returns
The given input `y` as PyTorch tensor. It has dimensions `[T, B, *out_shape]`, where `T` is the number of time steps (see attribute `max_num_ts_out`), `B` is the batch size and `out_shape` refers to the output feature shape, see `data.dataset.Dataset.out_shape`.

Return type
(torch.Tensor)

property permutation

Getter for attribute `permutation_`

Multilingual universal Dependencies Dataset

A data handler for the multilingual universal dependencies dataset:

https://universaldependencies.org/

This dataset is a Part-of-Speech tagging dataset that assigns to each token in a sentence one of a set of universal syntactic tags. We adapt this dataset to a Continual Learning scenario by considering Part-of-Speech tagging in different languages as different tasks.

class hypnettorch.data.timeseries.mud_data.**MUDData**(*task_data*, *vocabulary=None*, *tagset=None*)

Bases: *SequentialDataset*

Datahandler for the multilingual universal dependencies dataset.

Parameters

- **task_data** – A preprocessed dataset structure. Please use function *get_mud_handlers()* to create instances of this class.
- **vocabulary** (*list or tuple, optional*) – The vocabular, i.e., a list of words that allows us to decode input sentences.
- **tagset** (*list or tuple, optional*) – The PoS tagset.

decode_batch(*inputs*, *outputs*, *sample_ids=None*)

Decode a batch of input and output samples into strings.

This method translates a batch of input and output sequences (consisting of vocabulary and tagset indices) into actual sentences consisting of strings.

Note: This method is only applicable if `vocabulary` and `tagset` were provided to the constructor.

Parameters

- **inputs** (*numpy.ndarray or torch.Tensor*) – Input samples as provided to or returned from method *input_to_torch_tensor()*.
- **outputs** (*numpy.ndarray or torch.Tensor*) – Output samples as provided to or returned from method *output_to_torch_tensor()*.
- **sample_ids** (*numpy.ndarray*) – See method `train_ids_to_indices()`. If provided, the returned sentences are cropped to the actual sequence length.

Returns

Tuple containing:

- **in_words** (list): List of list of strings, where each string corresponds to a word in the corresponding input sentence of `inputs`.
- **out_tags** (list): List of list of strings, where each string corresponds to the output tag corresponding to the tag ID read from `outputs`.

Return type
(tuple)

`get_identifier()`

Returns the name of the dataset.

`input_to_torch_tensor(x, device, mode='inference', force_no_preprocessing=False, sample_ids=None)`

This method can be used to map the internal numpy arrays to PyTorch tensors.

Note: If `sample_ids` are provided, then padding will be reduced according to the sample within the minibatch with the longest sequence length.

Parameters
`(....)` – See docstring of method `data.dataset.Dataset.input_to_torch_tensor()`.

Returns
See docstring of method `data.sequential_dataset.SequentialDataset.input_to_torch_tensor()`.

Return type
(torch.LongTensor)

`output_to_torch_tensor(y, device, mode='inference', force_no_preprocessing=False, sample_ids=None)`

Identical to method `data.sequential_dataset.SequentialDataset.output_to_torch_tensor()`.

However, if `sample_ids` are provided, then the same padding behavior as elicited by method *input_to_torch_tensor()* is performed.

`hypnettorch.data.timeseries.mud_data.get_mud_handlers(data_path, num_tasks=5)`

This function instantiates `num_tasks` objects of the class *MUDData* each of which will contain a PoS dataset for a different language.

Parameters

- **data_path** (*str*) – See argument `data_path` of class `data.timeseries.smnist_data.SMNISTData`. If not existing, the dataset will be downloaded into this folder.
- **num_tasks** (*int, optional*) – The number of data handlers that should be returned by this function.

Returns
A list of data handlers, each corresponding to an object of class *MUDData* object.

Return type
(list)

Dataset for the Audioset task

A data handler for the audioset dataset taken from:

https://research.google.com/audioset/download.html

Data were preprocessed with the script `data.timeseries.structure_audioset` and then uploaded to dropbox. If this link becomes invalid, the data has to be preprocessed from scratch.

class hypnettorch.data.timeseries.audioset_data.**AudiosetData**(*data_path*, *use_one_hot=True*, *validation_size=0*, *target_per_timestep=True*, *rseed=None*)

Bases: *SequentialDataset*

Datahandler for the audioset task.

Parameters

- **data_path** (*str*) – Where should the dataset be read from? If not existing, the dataset will be downloaded into this folder.
- **use_one_hot** (*bool*) – Whether the class labels should be represented in a one-hot encoding.
- **validation_size** (*int*) – The number of validation samples.
- **target_per_timestep** (*bool, optional*) – If activated, the one-hot encoding of the current image will be copied across the entire sequence. Else, there is a single target for the entire sequence (rather than one per timestep.
- **rseed** (*int, optional*) – If `None`, the current random state of numpy is used to select a validation set from the training data. Otherwise, a new random state with the given seed is generated.

get_identifier()

Returns the name of the dataset.

Stroke MNIST (SMNIST) Dataset

A data handler for the stroke mnist data as discribed here:

https://github.com/edwin-de-jong/mnist-digits-stroke-sequence-data/

The data was preprocessed with the script `data.timeseries.preprocess_smnist` and then uploaded to dropbox. If this link becomes invalid, the data has to be preprocessed from scratch.

class hypnettorch.data.timeseries.smnist_data.**SMNISTData**(*data_path*, *use_one_hot=False*, *validation_size=0*, *target_per_timestep=True*)

Bases: *SequentialDataset*

Datahandler for stroke MNIST.

Note: That the outputs are always provided as one-hot encodings of duration equal to one. One can decide to make these targets span the entirety of the sequence (by repeating it over timesteps) by setting `target_per_timestep` to `True`.

Parameters

- **data_path** (*str*) – Where should the dataset be read from? If not existing, the dataset will be downloaded into this folder.
- **use_one_hot** (*bool*) – Whether the class labels should be represented in a one-hot encoding.
- **validation_size** (*int*) – The number of validation samples. Validation samples will be taking from the training set (the first n samples).
- **target_per_timestep** (*bool*) – If activated, the one-hot encoding of the current image will be copied across the entire sequence. Else, there is a single target for the entire sequence (rather than one per timestep.

get_identifier()

Returns the name of the dataset.

Custom Datasets

Dataset from random recurrent teacher networks

We consider a student-teacher setup. The dataset is meant for continual learning, such that an individual teacher (individual task) is used to determine the computation of a subspace of the activations of a recurrent student network.

This is a synthetic dataset that will allow the manual construction of the optimal student network that solves all tasks simultanously. As such, this student network can be compared to trained networks (either continually or in parallel on multiple tasks).

To be more precise, we set the teacher to be an Elman-type recurrent network (see `mnets.simple_rnn.SimpleRNN`):

$$\begin{aligned} r_t^{(k)} &= \sigma(A^{(k)} r_{t-1}^{(k)} + x_t) \\ s_t^{(k)} &= \sigma(B^{(k)} r_t^{(k)}) \\ t_t^{(k)} &= C^{(k)} s_t^{(k)} \end{aligned}$$

Where k is a unique task identifier (in the context of multiple teachers), x_t is the network input at time t, the recurrent state is initialized at zero $r_0^{(k)} = 0$ and $\sigma()$ is a user-defined non-linearity. The non-linear output computation $s_t^{(k)}$ is optional.

We assume an input $x_t \in \mathbb{R}^{n_{\text{in}}}$ and a target dimensionality of n_{out}. $A^{(k)} \in \mathbb{R}^{n_{\text{in}} \times n_{\text{in}}}$, $B^{(k)} \in \mathbb{R}^{n_{\text{in}} \times n_{\text{in}}}$ and $C^{(k)} \in \mathbb{R}^{n_{\text{out}} \times n_{\text{in}}}$ are random matrices that determine the teacher network's input-output mapping.

Having a task setup like this one can manually construct an RNN network that can solve multiple of such tasks to perfection (assuming a task-specific output head). For instance, consider the following Elman-type RNN with task-specific output head.

$$\begin{aligned} h_t &= \sigma(W_{hh} h_{t-1} + W_{ih} x_t + b_h) \\ o_t &= \sigma(W_{ho} h_t + b_o) \\ y_t^{(k)} &= W^{(k)} o_t + b^{(k)} \end{aligned}$$

With $h_t \in \mathbb{R}^{n_{\text{h}}}$ being the hidden state (we also assume $o_t \in \mathbb{R}^{n_{\text{h}}}$).

We can assign this network the following weights to ensure that all K tasks are solved to perfection:

- $b_h, b_o, b^{(k)} = 0$

- $W_{ih} = \begin{pmatrix} I \\ \vdots \\ I \\ O \end{pmatrix}$ where $I \in \mathbb{R}^{n_{\text{in}} \times n_{\text{in}}}$ refers to the identity matrix that simply copies the input into separate subspaces of the hidden state
- The hidden-to-hidden weights would be block diagonal:

$$W_{hh} = \begin{pmatrix} A^{(1)} & & & \\ & \ddots & & \\ & & A^{(K)} & \\ & & & O \end{pmatrix}$$

- The hidden-to-output weights would be block diagonal:

$$W_{ho} = \begin{pmatrix} B^{(1)} & & & \\ & \ddots & & \\ & & B^{(K)} & \\ & & & O \end{pmatrix}$$

- The task-specific output matrix would be

$$W^{(k)} = \begin{pmatrix} O & \dots & C^{(k)} & \dots & O \end{pmatrix}$$

class hypnettorch.data.timeseries.rnd_rec_teacher.**RndRecTeacher**(*num_train=1000*, *num_test=100*, *num_val=None*, *n_in=7*, *n_out=7*, *sigma='tanh'*, *mat_A=None*, *mat_B=None*, *mat_C=None*, *orth_A=False*, *rank_A=-1*, *max_sv_A=-1.0*, *no_extra_fc=False*, *inputs=None*, *input_range=(-1, 1)*, *n_ts_in=10*, *n_ts_out=-1*, *rseed=None*)

Bases: *SequentialDataset*

Create a dataset from a random recurrent teacher.

Parameters

- **num_train** (*int*) – Number of training samples.
- **num_test** (*int*) – Number of test samples.
- **num_val** (*int, optional*) – Number of validation samples.
- **n_in** (*int*) – Dimensionality of inputs x_t.
- **n_out** (*int*) – Dimensionality of outputs $y_t^{(k)}$.
- **sigma** (*str*) – Name of the nonlinearity $\sigma()$ to be used.
  - 'linear'
  - 'sigmoid'
  - 'tanh'

- **mat_A** (*numpy.ndarray, optional*) – A numpy array of shape `[n_in, n_in]` representing matrix $A^{(k)}$. If not specified, a random matrix will be generated.
- **mat_B** (*numpy.ndarray, optional*) – A numpy array of shape `[n_in, n_in]` representing matrix $B^{(k)}$. If not specified, a random matrix will be generated.
- **mat_C** (*numpy.ndarray, optional*) – A numpy array of shape `[n_out, n_in]` representing matrix $C^{(k)}$. If not specified, a random matrix will be generated.
- **orth_A** (*bool*) – If $A^{(k)}$ is randomly generated and this option is activated, then $A^{(k)}$ will be initialized as an orthogonal matrix.
- **rank_A** (*int, optional*) – The rank of the randomly generated matrix $A^{(k)}$. Note, this option is mutually exclusive with option `orth_A`.
- **max_sv_A** (*float, optional*) – The maximum singular value of the randomly generated matrix $A^{(k)}$. Note, this option is mutually exclusive with option `orth_A`.
- **no_extra_fc** – If `True`, the hidden fully-connected layer using matrix $B^{(k)}$ will be omitted when computed targets from the teacher. Hence, the teacher computation becomes:

$$r_t^{(k)} = \sigma(A^{(k)} r_{t-1}^{(k)} + x_t)$$
$$t_t^{(k)} = C^{(k)} r_t^{(k)}$$

- **inputs** (*numpy.ndarray, optional*) – The inputs x_t to be used. Has to be an array of shape `[n_ts_in, N, n_in]` with `N = num_train + num_test + (0 if num_val is None else num_val)`.
- **input_range** (*tuple*) – Tuple of integers. Used as ranges for a uniform distribution from which input samples x_t are drawn.
- **n_ts_in** (*int*) – The number of input timesteps.
- **n_ts_out** (*int, optional*) – The number of output timesteps. Can be greater than `n_ts_in`. In this case, the inputs at time greater than `n_ts_in` will be zero.
- **rseed** (*int, optional*) – If `None`, the current random state of numpy is used to generate the data. Otherwise, a new random state with the given seed is generated.

static construct_ideal_student(*net*, *dhandlers*)

Set the weights of an RNN such that it perfectly solves all tasks represented by the teachers in `dhandlers`.

Note: This method only works properly if the RNN `net` is properly setup such that its computation resembles the target computation of the individual teachers. I.e., an ideal student can be constructed by only modifying the weights.

Parameters

- **net** (*mnets.simple_rnn.SimpleRNN*) – The student RNN whose weights will be overwritten. Importantly, this method does not ensure that the teacher computation is compatible with the given student network.

  Note: The internal weights of the network are modified in-place.

- **dhandlers** (*list*) – List of datasets from teachers (i.e., instances of class *RndRecTeacher*). The RNN `net` must have at least as many output heads as `len(dhandlers)`.

get_identifier()

Returns the name of the dataset.

property mat_A

The teacher matrix $A^{(k)}$.

Type
numpy.ndarray

property mat_B

The teacher matrix $B^{(k)}$.

Type
numpy.ndarray

property mat_C

The teacher matrix $C^{(k)}$.

Type
numpy.ndarray

Continual Learning Datasets

Set of cognitive tasks

A data handler for cognitive tasks as implemented in Masse et al (PNAS). The user can construct individual datasets with this data handler and use each of these datasets to train a model in a continual leraning setting.

class hypnettorch.data.timeseries.cognitive_tasks.cognitive_data.**CognitiveTasks**(*task_id=0*, *num_train=80*, *num_test=20*, *num_val=None*, *rstate=None*)

Bases: *Dataset*

An instance of this class shall represent a one of the 20 cognitive tasks.

Generate a new dataset.

We use the MultiStimulus class from Masse el al. to genereate the inputs and outputs of different cognitive tasks in accordance with the data handling structures of the hnet code base.

Note that masks (part of the Masse et al. trial generator) will be handled independently of this data handler.

Parameters

- **num_train** (*int*) – Number of training samples.
- **num_test** (*int*) – Number of test samples.
- **num_val** (*optional*) – Number of validation samples.
- **rstate** – If None, the current random state of numpy is used to generate the data.

get_identifier()

Returns the name of the dataset.

input_to_torch_tensor(*x*, *device*, *mode='inference'*, *force_no_preprocessing=False*, *sample_ids=None*)

This method can be used to map the internal numpy arrays to PyTorch tensors.

Parameters

(....) – See docstring of method `data.dataset.Dataset.input_to_torch_tensor()`.

Returns

The given input `x` as 3D PyTorch tensor. It has dimensions `[T, B, N]`, where `T` is the number of time steps per stimulus, `B` is the batch size and `N` the number of input units.

Return type

(torch.Tensor)

output_to_torch_tensor(*y*, *device*, *mode='inference'*, *force_no_preprocessing=False*, *sample_ids=None*)

Similar to method *input_to_torch_tensor()*, just for dataset outputs.

Parameters

(....) – See docstring of method `data.dataset.Dataset.output_to_torch_tensor()`.

Returns

A tensor of shape `[T, B, C]`, where `T` is the number of time steps per stimulus, `B` is the batch size and `C` the number of classes.

Return type

(torch.Tensor)

Sequence of Stroke MNIST Samples (SeqSMNIST) Dataset

A data handler to generate a set of sequential stroke MNIST tasks for continual learning. The used stroke MNIST data was already preprocessed with the script `data.timeseries.preprocess_smnist` (see also the corresponding data handler in `data.timeseries.smnist_data`).

The task

Given a sequence of two smnist digits of length `n` (e.g. `2,5,5,2,2` with `n=5`), classify which of the `2**n` possible binary sequences (classes) the presented sequence belongs to. E.g., for `n=3` the number of classes would be 8 (corresponding to all possible sequences with two digits (`0` and `1` here): `000, 001, 010, 100, 011, 110, 101, 111`.

The individual tasks of the task family differ in which digits are used to generate the binary sequences. Considering all possible pairs of digits we can generate (10**2-10) / 2 = 45 tasks.

class hypnettorch.data.timeseries.seq_smnist.**SeqSMNIST**(*data_path*, *use_one_hot=True*, *num_train=1600*, *num_test=400*, *num_val=0*, *target_per_timestep=True*, *sequence_length=4*, *digits=(0, 1)*, *two_class=False*, *upsample_control=False*, *fix_class_partition=False*, *rseed=None*)

Bases: *SequentialDataset*

Datahandler for one sequential stroke MNIST task (as described above).

Note: That the outputs are always provided as one-hot encodings of duration equal to one. One can decide to make these targets span the entirety of the sequence (by repeating it over timesteps) by setting `target_per_timestep` to `True`.

Parameters

- **data_path** (*str*) – Where should the dataset be read from? If not existing, the dataset will be downloaded into this folder.
- **use_one_hot** (*bool*) – Whether the class labels should be represented in a one-hot encoding.
- **num_train** (*int*) – Number of training samples to be generated.
- **num_test** (*int*) – Number of test samples to be generated.
- **num_val** (*int*) – Number of validation samples to be generated.
- **target_per_timestep** (*bool*) – If activated, the one-hot encoding of the current image will be copied across the entire sequence. Else, there is a single target for the entire sequence (rather than one per timestep.
- **sequence_length** (*int*) – The length of the binary sequence to be classified. This also affects the number of classes which is `2**n`.
- **digits** (*tuple*) – The two digits that shall be used for generating the binary sequence.
- **two_class** (*bool*) – When true, instead of classifying each possible sequence individually, sequences are randomly grouped into two classes. This makes the number of classes (and therefore the chance level) independent of the sequence length.
- **upsample_control** (*bool*) – If `True`, instead of building sequences of digits, we upsample single digits by a factor given by `seq_len`.
- **fix_class_partition** (*bool*) – TODO
- **rseed** (*int*) – Seed for numpy random state.

`get_identifier()`

Returns the name of the dataset.

Split Audioset Dataset

The module `data.timeseries.split_audioset` contains a wrapper for data handlers for the SplitAudioset task. It is based on the module `data.special.split_mnist`.

`class hypnettorch.data.timeseries.split_audioset.SplitAudioset(`*data_path*, *use_one_hot=True*, *validation_size=1000*, *target_per_timestep=True*, *rseed=None*, *labels=[0, 1]*, *full_out_dim=False*`)`

Bases: *AudiosetData*

An instance of the class shall represent a SplitAudioset task.

Parameters

- **(....)** – See docstring of class `data.timeseries.audioset_data.AudiosetData`.
- **validation_size** (*int*) – The size of the validation set of each individual data handler.
- **labels** (*list*) – The labels that should be part of this task.
- **full_out_dim** (*bool*) – Choose the original Audioset labels instead of the new task output dimension. This option will affect the attributes `data.dataset.Dataset.num_classes` and `data.dataset.Dataset.out_shape`.

get_identifier()

Returns the name of the dataset.

transform_outputs(*outputs*)

Transform the outputs from the 100D Audioset dataset into proper labels based on the constructor argument `labels`.

I.e., the output will have `len(labels)` classes.

Example

Split with labels [2,3]

1-hot encodings: [0,0,0,1,...,0,0,0,0,0,0] -> [0,1]

labels: 3 -> 1

Parameters
outputs – 2D numpy array of outputs.

Returns
2D numpy array of transformed outputs.

hypnettorch.data.timeseries.split_audioset.**get_split_audioset_handlers**(*data_path*, *use_one_hot=True*, *validation_size=0*, *target_per_timestep=True*, *num_classes_per_task=10*, *num_tasks=5*, *rseed=None*)

This function instantiates `num_tasks` objects of the class `AudiosetData` which will contain a disjoint set of labels.

The SplitAudioset task consists of `num_tasks` tasks which consist of a classification problem with `num_classes_per_task` classes from our preprocessed Audioset data set.

Parameters

- **(....)** – See docstring of class `data.timeseries.audioset_data.AudiosetData`.
- **validation_size** (*int*) – The size of the validation set of each individual data handler.
- **num_classes_per_task** (*int*) – Number of classes to put into one data handler. If `2`, then every data handler will include 2 classes.
- **num_tasks** (*int*) – The number of data handlers that should be returned by this function.
- **rseed** (*int, optional*) – The `rseed` is passed when constructing instances of class *SplitAudioset*. In addition, it is used to shuffle the classes before splitting Audioset into tasks.

Returns
A list of data handlers, each corresponding to a *SplitAudioset* object.

Return type
(list)

Split SMNIST Dataset

The module `data.timeseries.split_smnist` contains a wrapper for data handlers for a set of SplitSMNIST tasks (a partitioning of classes from the `data.timeseries.smnist_data.SMNISTData` dataset). The implementation is based on the module `data.special.split_mnist`.

class hypnettorch.data.timeseries.split_smnist.**SplitSMNIST**(*data_path*, *use_one_hot=False*, *validation_size=1000*, *target_per_timestep=True*, *labels=[0, 1]*, *full_out_dim=False*)

Bases: *SMNISTData*

An instance of the class shall represent a SplitSMNIST task.

Parameters

- **data_path** (*str*) – See argument `data_path` of class `data.timeseries.smnist_data.SMNISTData`.
- **use_one_hot** (*bool*) – Whether the class labels should be represented in a one-hot encoding.
- **validation_size** (*int*) – The number of validation samples. Validation samples will be taken from the training set (the first n samples).
- **target_per_timestep** (*str*) – See argument `target_per_timestep` of class `data.timeseries.smnist_data.SMNISTData`.
- **labels** (*list*) – The labels that should be part of this task.
- **full_out_dim** (*bool*) – Choose the original SMNIST instead of the new task output dimension. This option will affect the attributes `data.dataset.Dataset.num_classes` and `data.dataset.Dataset.out_shape`.

get_identifier()

Returns the name of the dataset.

transform_outputs(*outputs*)

Transform the outputs from the 10D MNIST dataset into proper labels based on the constructor argument `labels`.

I.e., the output will have `len(labels)` classes.

Example

Split with labels [2,3]

1-hot encodings: [0,0,0,1,0,0,0,0,0,0] -> [0,1]

labels: 3 -> 1

Parameters
outputs – 2D numpy array of outputs.

Returns
2D numpy array of transformed outputs.

hypnettorch.data.timeseries.split_smnist.**get_split_smnist_handlers**(*data_path*, *use_one_hot=True*, *validation_size=0*, *target_per_timestep=True*, *num_classes_per_task=2*, *num_tasks=None*)

This function instantiates 5 objects of the class *SplitSMNIST* which will contain a disjoint set of labels.

The SplitSMNIST task consists of 5 tasks corresponding to stroke trajectories for the images with labels [0,1], [2,3], [4,5], [6,7], [8,9].

Parameters

- **data_path** (*str*) – See argument data_path of class data.timeseries.smnist_data.SMNISTData.
- **use_one_hot** (*bool*) – Whether the class labels should be represented in a one-hot encoding.
- **validation_size** (*int*) – The size of the validation set of each individual data handler.
- **target_per_timestep** (*str*) – See argument target_per_timestep of class data.timeseries.smnist_data.SMNISTData.
- **num_classes_per_task** (*int*) – Number of classes to put into one data handler. If 2, then every data handler will include 2 digits.
- **num_tasks** (*int, optional*) – The number of data handlers that should be returned by this function.

Returns

A list of data handlers, each corresponding to a *SplitSMNIST* object.

Return type
(list)

See documentation of subpackages *special* and *timeseries*.

CHAPTER

TWO

HYPERNETWORKS

Contents

A hypernetwork is a neural network that produces the weights of another network. As such, it can be seen as a specific type of main network (aka neural network). Therefore, each hypernetwork has a specific interface `hypnettorch.hnets.hnet_interface.HyperNetInterface` which is derived from the main network interface `hypnettorch.mnets.mnet_interface.MainNetInterface`.

Note: All hypernetworks in this subpackage implement the abstract interface `hypnettorch.hnets.hnet_interface.HyperNetInterface` to provide a consistent interface for users.

2.1 Hypernetwork Interface

The module `hypnettorch.hnets.hnet_interface` contains an interface for hypernetworks.

A hypernetworks is a special type of neural network that produces the weights of another neural network (called the main or target networks, see `hypnettorch.mnets.mnet_interface`). The name "hypernetworks" was introduced in

> *Ha et al., "Hypernetworks", 2016. <https://arxiv.org/abs/1609.09106>*

The interface ensures that we can consistently use different types of these networks without knowing their specific implementation details (as long as we only use functionalities defined in class *HyperNetInterface*).

class hypnettorch.hnets.hnet_interface.**HyperNetInterface**

Bases: *MainNetInterface*

A general interface for hypernetworks.

add_to_uncond_params(*dparams*, *params=None*)

Add perturbations to unconditional parameters.

This method simply adds a perturbation `dparams` ($d\theta$) to the unconditional parameters θ.

Parameters

- **dparams** (*list*) – List of tensors.
- **params** (*list, optional*) – List of tensors. If unspecified, attribute *unconditional_params* is taken instead. Otherwise, the method simply returns `params + dparams`.

Returns
List were elements of `dparams` and unconditional params (or `params`) are summed together.

Return type
(list)

property conditional_param_shapes

A list of lists of integers denoting the shape of every parameter tensor belonging to the *conditional* parameters associated with this hypernetwork (i.e., the complement of those returned by *unconditional_param_shapes*). Note, the returned list is a subset of the shapes maintained in *hypnettorch.mnets.mnet_interface.MainNetInterface.param_shapes* and is independent whether these parameters are internally maintained (i.e., occuring within *conditional_params*).

Type
list

property conditional_param_shapes_ref

A list of integers that has the same length as *conditional_param_shapes*. Each entry represents an index within attribute *hypnettorch.mnets.mnet_interface.MainNetInterface.param_shapes*.

It can be used to gain access to meta information about conditional parameters via attribute *hypnettorch.mnets.mnet_interface.MainNetInterface.param_shapes_meta*.

Type
list

property conditional_params

The complement of the internally maintained parameters hold by attribute *unconditional_params*.

A typical example of these parameters are embedding vectors. In continual learning, for instance, there could be a separate task- embedding per task used as hypernet input, see

von Oswald et al., "Continual learning with hypernetworks", ICLR 2020. https://arxiv.org/abs/1906.00695

Note: This attribute is `None` if there are no conditional parameters that are internally maintained.

Type
list or None

convert_out_format(*hnet_out*, *src_format*, *trgt_format*)

Convert the hypernetwork output into another format.

This is a helper method to easily convert the output of a hypernetwork into different formats. Cf. argument `ret_format` of method *forward()*.

Parameters

- **hnet_out** (*list or torch.Tensor*) – See return value of method *forward()*.
- **src_format** (*str*) – The format of argument `hnet_out`. See argument `ret_format` of method *forward()*.
- **trgt_format** (*str*) – The target format in which `hnet_out` should be converted. See argument `ret_format` of method *forward()*.

Returns

The input `hnet_out` converted into the target format `trgt_format`.

Return type
(list or torch.Tensor)

abstract forward(*uncond_input=None*, *cond_input=None*, *cond_id=None*, *weights=None*, *distilled_params=None*, *condition=None*, *ret_format='squeezed'*)

Perform a pass through the hypernetwork.

Parameters

- **uncond_input** (*optional*) – The unconditional input to the hypernetwork.

  Note: Not all scenarios require a hypernetwork with unconditional inputs. For instance, a task-conditioned hypernetwork only receives a task-embedding (a conditional input) as input.

- **cond_input** (*optional*) – If applicable, the conditional input to the hypernetwork.
- **cond_id** (*int or list, optional*) – The ID of the condition to be applied. Only applicable if conditional inputs/weights are maintained internally and conditions are discrete.

  Can also be a list of IDs if a batch of weights should be produced.

  Condition IDs have to be between 0 and `num_conditions`.

  Note: Option is mutually exclusive with option `cond_input`.

- **weights** (*list or dict, optional*) – List of weight tensors, that are used as hypernetwork parameters. If not all weights are internally maintained, then this argument is non-optional.

  If a `list` is provided, then it either has to match the length of *hypnettorch.mnets.mnet_interface.MainNetInterface.hyper_shapes_learned* (if specified) or the length of attribute *hypnettorch.mnets.mnet_interface.MainNetInterface.param_shapes*.

  If a `dict` is provided, it must have at least one of the following keys specified: - `'uncond_weights'` (list): Contains unconditional weights. - `'cond_weights'` (list): Contains conditional weights.

- **distilled_params** (*optional*) – See docstring of method *hypnettorch.mnets.mnet_interface.MainNetInterface.forward()*.
- **condition** (*optional*) – See docstring of method *hypnettorch.mnets.mnet_interface.MainNetInterface.forward()*.
- **ret_format** (*str*) – The format in which the generated weights are returned. The following options are available.
  - `'flattened'`: The hypernet output will be a tensor of shape `[batch_size, num_outputs]` (see *num_outputs*).
  - `'sequential'`: A list of length *batch size* is returned that contains lists of length `len(target_shapes)`, which contain tensors with shapes determined by attribute *target_shapes*. Hence, each entry of the returned list contains the weights for one sample in the input batch.
  - `'squeezed'`: Same as `'sequential'`, but if the batch size is `1`, the list will be unpacked, such that a list of tensors is returned (rather than a list of list of tensors).

  #### Example

  Assume *target_shapes* to be `[[10, 5], [10]]` and `cond_input` to be the only input to the hypernetwork, which is a batch of embeddings `[B, E]`, where `B` is the batch size and `E` is the embedding size.

  Note, `num_outputs = 60` in this case (cmp. *num_outputs*).

  If `'flattened'` is used, a tensor of shape `[B, 60]` is returned. If `'sequential'` or `'squeezed'` is used and `B > 1` (e.g., `B=3`), then a list of lists of tensors (here denoted by their shapes) is returned `[[[10, 5], [10]], [[10, 5], [10]], [[10, 5], [10]]]`. However, if `B == 1` and `'squeezed'` is used, then a list of tensors is returned, e.g., `[[10, 5], [10]]`.

Returns
See description of argument `ret_format`.

Return type
(list or torch.Tensor)

`get_task_emb`(*task_id*)

Returns the `task_id`-th element from attribute *conditional_params*.

Deprecated since version 1.0: Please access elements of attribute *conditional_params* directly, as the conditional parameters do not have to correspond to task embeddings.

Parameters
task_id (*int*) – Determines which element of *conditional_params* should be returned.

Returns
(torch.nn.Parameter)

`get_task_embs`()

Returns attribute *conditional_params*.

Deprecated since version 1.0: Please access attribute *conditional_params* directly, as the conditional parameters do not have to correspond to task embeddings.

Returns
(list or None)

property num_known_conds

The number of conditions known to this hypernetwork. If the number of conditions is discrete and internally maintained by the hypernetwork, then this attribute specifies how many conditions the hypernet manages.

Note: The option does not have to agree with the length of attribute *conditional_params*. For instance, in certain cases there are multiple conditional weights maintained per condition.

Type
int

property num_outputs

The total number of output neurons (number of weights generated for the target network). This quantity can be computed based on attribute *target_shapes*.

Type
int

property target_shapes

A list of list of integers representing the shapes of weight tensors generated, i.e., the hypernet output, which could be, for instance, the `mnets.mnet_interface.MainNetInterface.hyper_shapes_learned` of another network whose weights this hypernetwork is producing.

Type
list

property unconditional_param_shapes

A list of lists of integers denoting the shape of every parameter tensor belonging to the *unconditional* parameters associated with this hypernetwork. Note, the returned list is a subset of the shapes maintained in *hypnettorch.mnets.mnet_interface.MainNetInterface.param_shapes* and is independent whether these parameters are internally maintained (i.e., occuring within *unconditional_params*).

Type
list

property unconditional_param_shapes_ref

A list of integers that has the same length as *unconditional_param_shapes*. Each entry represents an index within attribute *hypnettorch.mnets.mnet_interface.MainNetInterface.param_shapes*.

Type
list

property unconditional_params

Internally maintained parameters of the hypernetwork **excluding** parameters that may be specific to a given condition, e.g., task embeddings in continual learning.

Hence, it is the portion of parameter tensors from attribute `mnets.mnet_interface.MainNetInterface.internal_params` that is not specific to a certain task/condition.

Note: This attribute is `None` if there are no unconditional parameters that are internally maintained.

Example

An example use-case for a hypernetwork h could be the following: $h(x, e_i; \theta)$, where x is an arbitrary input, e_i is a learned embedding (condition) and θ are the internal "unconditional" parameters of the hypernetwork. In some cases (for simplicity), the conditions e_i as well as the parameters θ are maintained internally by this class. This attribute can be used to gain access to the "unconditional" parameters θ, while `mnets.mnet_interface.MainNetInterface.internal_params` would return all "conditional" parameters e_i as well as the "unconditional" parameters θ.

Type
list or None

property unconditional_params_ref

A list of integers that has the same length as *unconditional_params*. Each entry represents an index within attribute *hypnettorch.mnets.mnet_interface.MainNetInterface.internal_params*.

If *unconditional_params* is `None`, the this attribute is `None` as well.

Example

Using an instance `hnet` that implements this interface, the following is `True`.

```
hnet.internal_params[hnet.unconditional_params_ref[i]] is hnet.
↪unconditional_params[i]
```

Note: This attribute has different semantics compared to *unconditional_param_shapes_ref* which points to locations within *hypnettorch.mnets.mnet_interface.MainNetInterface.param_shapes*, wheras this attribute points to locations within *hypnettorch.mnets.mnet_interface.MainNetInterface.internal_params*.

Type
list or None

2.2 Chunked Deconvolutional Hypernetwork with Self-Attention Layers

The module `hnets.chunked_deconv_hnet` implements a chunked version of the transpose convolutional hypernetwork represented by class `hnets.deconv_hnet.HDeconv` (similar as to `hnets.chunked_mlp_hnet.ChunkedHMLP` represents a chunked version of the full hypernetwork `hnets.mlp_hnet.HMLP`).

Therefore, an instance of class *ChunkedHDeconv* manages internally an instance of class `hnets.deconv_hnet.HDeconv`, which is invoked multiple time with a different additional input (the so called *chunk embedding*) to produce a chunk of the target weights at a time, which are later put together. See description of module `hnets.chunked_mlp_hnet` for more details.

Note: This type of hypernetwork is completely agnostic to the architecture of the target network. The splits happen at arbitrary locations in the flattened target network weight vector.

class hypnettorch.hnets.chunked_deconv_hnet.**ChunkedHDeconv**(*target_shapes*, *hyper_img_shape*, *chunk_emb_size=8*, *cond_chunk_embs=False*, *uncond_in_size=0*, *cond_in_size=8*, *num_layers=5*, *num_filters=None*, *kernel_size=5*, *sa_units=(1, 3)*, *verbose=True*, *activation_fn=ReLU()*, *use_bias=True*, *no_uncond_weights=False*, *no_cond_weights=False*, *num_cond_embs=1*, *use_spectral_norm=False*, *use_batch_norm=False*)

Bases: Module, *HyperNetInterface*

Implementation of a *chunked deconvolutional hypernet*.

The `target_shapes` will be flattened and split into chunks of size `chunk_size = np.prod(hyper_img_shape)`. In total, there will be `np.ceil(self.num_outputs/chunk_size)` chunks, where the last chunk produced might contain a remainder that is discarded.

Each chunk has it's own *chunk embedding* that is fed into the underlying hypernetwork.

Note: It is possible to set `uncond_in_size` and `cond_in_size` to zero if `cond_chunk_embs` is `True`.

(....)

See attributes of class `hnets.chunked_mlp_hnet.ChunkedHMLP`.

Parameters

- **(....)** – See constructor arguments of class `hnets.deconv_hnet.HDeconv`.
- **chunk_emb_size** (*int*) – See constructor arguments of class `hnets.chunked_mlp_hnet.ChunkedHMLP`.
- **cond_chunk_embs** (*bool*) – See constructor arguments of class `hnets.chunked_mlp_hnet.ChunkedHMLP`.

Initializes internal Module state, shared by both nn.Module and ScriptModule.

property chunk_emb_size

Getter for read-only attribute *chunk_emb_size*.

property cond_chunk_embs

Getter for read-only attribute *cond_chunk_embs*.

forward(*uncond_input=None*, *cond_input=None*, *cond_id=None*, *weights=None*, *distilled_params=None*, *condition=None*, *ret_format='squeezed'*)

Compute the weights of a target network.

Parameters

(....) – See docstring of method `hnets.chunked_mlp_hnet.ChunkedHMLP.forward()`.

Returns

See docstring of method `hnets.hnet_interface.HyperNetInterface.forward()`.

Return type
(list or torch.Tensor)

`get_chunk_emb`(*chunk_id=None*, *cond_id=None*)

Get the `chunk_id`-th chunk embedding.

Parameters
(....) – See docstring of method `hnets.chunked_mlp_hnet.ChunkedHMLP.get_chunk_emb()`.

Returns
(torch.nn.Parameter)

`get_cond_in_emb`(*cond_id*)

Get the `cond_id`-th (conditional) input embedding.

Parameters
(....) – See docstring of method `hnets.deconv_hnet.HDeconv.get_cond_in_emb()`.

Returns
(torch.nn.Parameter)

`property num_chunks`

Getter for read-only attribute *num_chunks*.

`training: bool`

2.3 Chunked MLP - Hypernetwork

The module `hnets.chunked_mlp_hnet` contains a *Chunked Hypernetwork*, that uses a full hypernetwork (see `hnets.mlp_hnet.HMLP`) to produce one chunk of the output weights at a time.

The hypernetwork $h_\theta(e)$ (with input e) operates as follows. The target outputs (see `hnets.hnet_interface.HyperNetInterface.target_shapes`) are flattened and split into equally sized chunks. Those chunks are separately generated by an internal full hypernetwork $h'_{\theta'}(e, c)$ (that is hidden from the user), where c denotes the chunk embedding, which are internally maintained and chunk-specific.

Note: This type of hypernetwork is completely agnostic to the architecture of the target network. The splits happen at arbitrary locations in the flattened target network weight vector.

`class hypnettorch.hnets.chunked_mlp_hnet.ChunkedHMLP`(*target_shapes*, *chunk_size*, *chunk_emb_size=8*, *cond_chunk_embs=False*, *uncond_in_size=0*, *cond_in_size=8*, *layers=(100, 100)*, *verbose=True*, *activation_fn=ReLU()*, *use_bias=True*, *no_uncond_weights=False*, *no_cond_weights=False*, *num_cond_embs=1*, *dropout_rate=-1*, *use_spectral_norm=False*, *use_batch_norm=False*)

Bases: `Module`, *HyperNetInterface*

Implementation of a *chunked fully-connected hypernet*.

The `target_shapes` will be flattened and split into chunks of size `chunk_size`. In total, there will be `np.ceil(self.num_outputs/chunk_size)` chunks, where the last chunk produced might contain a remainder that is discarded.

Each chunk has it's own *chunk embedding* that is fed into the underlying hypernetwork.

Note: It is possible to set `uncond_in_size` and `cond_in_size` to zero if `cond_chunk_embs` is `True`.

Parameters

- **`(....)`** – See constructor arguments of class `hnets.mlp_hnet.HMLP`.
- **`chunk_size`** (*int*) – The chunk size, i.e, the number of weights produced by individual forward passes of the internally maintained instance of a full hypernet (see `hnets.mlp_hnet.HMLP`) upon receiving a chunk embedding).
- **`chunk_emb_size`** (*int*) – The size of a chunk embedding.
- **`cond_chunk_embs`** (*bool*) – Whether chunk embeddings are unconditional (`False`) or conditional (`True`) parameters. See constructor argument `cond_chunk_embs`.

  Note: Embeddings will be initialized with a normal distribution using zero mean and unit variance.

- **`cond_chunk_embs`** – Consider chunk embeddings to be conditional. In this case, there will be a different set of chunk embeddings per condition (specified via `num_cond_embs`).

  If `False`, there will be a total of *num_chunks* chunk embeddings that are maintained within `hnets.hnet_interface.HyperNetInterface.unconditional_param_shapes`.
  If `True`, there will be `num_cond_embs * self.num_chunks` chunk embeddings that are maintained within `hnets.hnet_interface.HyperNetInterface.conditional_param_shapes`. However, if `num_cond_embs == 0`, then chunk embeddings have to be provided in a special way to the *forward()* method (see the corresponding argument `weights`).

Initializes internal Module state, shared by both nn.Module and ScriptModule.

`apply_chunked_hyperfan_init`(*method='in'*, *use_xavier=False*, *uncond_var=1.0*, *cond_var=1.0*, *eps=1e-05*, *cemb_normal_init=False*, *mnet=None*, *target_vars=None*)

Initialize the network using a chunked hyperfan init.

Inspired by the method Hyperfan Init which we implemented for the MLP hypernetwork in method `hnets.mlp_hnet.HMLP.apply_hyperfan_init()`, we heuristically developed a better initialization method for chunked hypernetworks.

Unfortunately, the *Hyperfan Init* method from the paper does not apply to this kind of hypernetwork, since we reuse the same hypernet output head for the whole main network.

Luckily, we can provide a simple heuristic. Similar to Meyerson & Miikkulainen we play with the variance of the input embeddings to affect the variance of the output weights.

In a chunked hypernetwork, the input for each chunk is identical except for the chunk embeddings $\mathbf{c}$. Let $\mathbf{e}$ denote the remaining inputs to the hypernetwork, which are identical for all chunks. Then, assuming the hypernetwork was initialized via fan-in init, the variance of the hypernetwork output $\mathbf{v}$ can be written as follows (see documentation of method `hnets.mlp_hnet.HMLP.apply_hyperfan_init()`):

$$\text{Var}(v) = \frac{n_e}{n_e + n_c}\text{Var}(e) + \frac{n_c}{n_e + n_c}\text{Var}(c)$$

Hence, we can achieve a desired output variance $\text{Var}(v)$ by initializing the chunk embeddings $\mathbf{c}$ via the following variance:

$$\text{Var}(c) = \max\Big\{0,\ \frac{1}{n_c}\big[(n_e + n_c)\text{Var}(v) - \ n_e\text{Var}(e)\big]\Big\}$$

Now, one important question remains. How do we pick a desired output variance $\text{Var}(v)$ for a chunk?

Note, a chunk may include weights from several layers. The likelihood for this to happen depends on the main net architecture and the chunk size (see constructor argument `chunk_size`). The smaller the chunk size, the less likely it is that a chunk will contain elements from multiple main net weight tensors.

In case each chunk would contain only weights from one main net weight tensor, we could simply pick the variance $\text{Var}(v)$ that would have been chosen by a main net initialization method (such as Xavier).

In case a chunk contains contributions from several main net weight tensors, we apply the following heuristic. If a chunk contains contributions of a set of main network weight tensors $W_1, \ldots, W_K$ with relative contribution sizes$n_1, \ldots, n_K$ such that $n_1 + \cdots + n_K = n_v$ where n_v denotes the chunk size and if the corresponding main network initialization method would require init variances $\text{Var}(w_1), \ldots, \text{Var}(w_K)$, then we simply request a weighted average as follow:

$$\text{Var}(v) = \frac{1}{n_v} \sum_{k=1}^{K} n_k \text{Var}(w_k)$$

What about bias vectors? Usually, the variance analysis applied to Xavier or Kaiming init assumes that biases are initialized to zero. This is not possible in this setting, as it would require assigning a negative variance to $\mathbf{c}$. Instead, we follow the default PyTorch initialization (e.g., see method `reset_parameters` in class `torch.nn.Linear`). There, bias vectors are initialized uniformly within a range of $\pm\frac{1}{\sqrt{f_{\text{in}}}}$ where f_{in} refers to the fan-in of the layer. This type of initialization corresponds to a variance of $\text{Var}(v) = \frac{1}{3f_{\text{in}}}$.

Note: All hypernet inputs are assumed to be zero-mean random variables.

Note: To avoid that the variances with which chunks are initialized have to be clipped (because they are too small or even negative), the variance of the remaining hypernet inputs should be properly scaled. In general, one should adhere the following rule

$$\text{Var}(e) < \frac{n_e + n_c}{n_e}\text{Var}(v)$$

This method will calculate and print the maximum value that should be chosen for $\text{Var}(e)$ and will print warnings if variances have to be clipped.

Parameters

- **(....)** – See arguments of method `hnets.mlp_hnet.HMLP.apply_hyperfan_init()`.
- **method** (*str*) – The type of initialization that should be applied. Possible options are:
  - `in`: Use *Chunked Hyperfan-in*, i.e., rather the output variances of the hypernetwork should correspond to fan-in variances.
  - `out`: Use *Chunked Hyperfan-out*, i.e., rather the output variances of the hypernetwork should correspond to fan-out variances.
  - `harmonic`: Use the harmonic mean of the fan-in and fan-out variance as target variance of the hypernetwork output.

- **eps** (*float*) – The minimum variance with which a chunk embedding is initialized.
- **cemb_normal_init** (*bool*) – Use normal init for chunk embeddings rather than uniform init.
- **target_vars** (*list or dict, optional*) – The variance of the distribution for each parameter tensor generated by this hypernetwork. Target variance values can either be provided as list of length `len(hnet.target_shapes)` or as dictionary. The usage is analoguous to the usage of parameter `w_val` of method `hnets.mlp_hnet.HMLP.apply_hyperfan_init()`.

Note: This method currently does not allow initial output distributions with non-zero mean. However, the docstring of method `probabilistic.gauss_hnet_init.gauss_hyperfan_init()` describes how this is in principle feasible and might be incorporated in the future.

Note: Unspecified target variances for parameter tensors of type `'weight'` or `'bias'` are computed as described above. Default target variances for all other parameter tensor types are simply 1.

---

property chunk_emb_size

See constructor argument `chunk_emb_size`.

property cond_chunk_embs

See constructor argument `cond_chunk_embs`.

distillation_targets()

Targets to be distilled after training.

See docstring of abstract super method `mnets.mnet_interface.MainNetInterface.distillation_targets()`.

Returns
See `hnets.mlp_hnet.HMLP.distillation_targets()`.

forward(*uncond_input=None*, *cond_input=None*, *cond_id=None*, *weights=None*, *distilled_params=None*, *condition=None*, *ret_format='squeezed'*)

Compute the weights of a target network.

Parameters

- **(....)** – See docstring of method `hnets.mlp_hnet.HMLP.forward()`.
- **weights** (*list or dict, optional*) – If provided as `dict` and chunk embeddings are considered conditional (see constructor argument `cond_chunk_embs`), then the additional key `chunk_embs` can be used to pass a batch of chunk embeddings. This option is mutually exclusive with the option of passing `cond_id`. Note, if conditional inputs via `cond_input` are expected, then the batch sizes must agree.

  A batch of chunk embeddings is expected to be tensor of shape `[B, num_chunks, chunk_emb_size]`, where `B` denotes the batch size.

Returns
See docstring of method `hnets.hnet_interface.HyperNetInterface.forward()`.

Return type
(list or torch.Tensor)

get_chunk_emb(*chunk_id=None*, *cond_id=None*)

Get the chunk_id-th chunk embedding.

Parameters

- **chunk_id** (*int, optional*) – A number between 0 and *num_chunks* - 1. If not specified, a full chunk matrix with shape [num_chunks, chunk_emb_size] is returned. Otherwise, the chunk_id-th row is returned.
- **cond_id** (*int*) – Is mandatory if constructor argument cond_chunk_embs was set. Determines the set of chunk embeddings to be considered.

Returns
(torch.nn.Parameter)

get_cond_in_emb(*cond_id*)

Get the cond_id-th (conditional) input embedding.

Parameters
(....) – See docstring of method hnets.mlp_hnet.HMLP.get_cond_in_emb().

Returns
(torch.nn.Parameter)

property num_chunks

The number of chunks that make up the final hypernet output.

This also corresponds to the number of chunk embeddings required per forward sweep.

Type
int

training: bool

2.4 Deconvolutional Hypernetwork with Self-Attention Layers

The module hnets.deconv_hnet implements a hypernetwork that uses transpose convolutions (like the generator of a GAN) to generate weights. Though, as convolutions usually suffer from only capturing local correlations sufficiently, we incorporate the self-attention mechanism developed by

Zhang et al., Self-Attention Generative Adversarial Networks, 2018.

See utils.self_attention_layer.SelfAttnLayerV2 for details on this layer type.

The purpose of this network can be seen as the convolutional analogue of the fully-connected hnets.mlp_hnet.HMLP. Hence, it produces all weights in one go; and does not utilize chunking to obtain better weight compression ratios (a chunked version can be found in module hnets.chunked_deconv_hnet).

class hypnettorch.hnets.deconv_hnet.**HDeconv**(*target_shapes*, *hyper_img_shape*, *uncond_in_size=0*, *cond_in_size=8*, *num_layers=5*, *num_filters=None*, *kernel_size=5*, *sa_units=(1, 3)*, *verbose=True*, *activation_fn=ReLU()*, *use_bias=True*, *no_uncond_weights=False*, *no_cond_weights=False*, *num_cond_embs=1*, *use_spectral_norm=False*, *use_batch_norm=False*)

Bases: Module, *HyperNetInterface*

Implementation of a *deconvolutional full hypernet*.

This is a convolutional network, employing transpose convolutions. The network structure is inspired by the DCGAN generator structure, though, we are additionally using self-attention layers to model global dependencies.

In general, each transpose convolutional layer will roughly double its input size. Though, we set the hard constraint that if the input size of a transpose convolutional layer would be smaller 4, then it doesn't change the size.

The network allows to maintain a set of embeddings internally that can be used as conditional input (cmp. `hnets.mlp_hnet.HMLP`).

Parameters

- **`(....)`** – See constructor arguments of class `hnets.mlp_hnet.HMLP`.
- **`hyper_img_shape`** (*tuple*) – Since the network has a (de-)convolutional output layer, the output will be in an image-like shape. Therefore, it won't be possible to precisely produce the number of weights prescribed by `target_shapes`. Therefore, the *hyper-image* size defined via this option has to be chosen big enough, i.e., the number of pixels must be greater equal than the number of weights to be produced. Remaining pixels will be discarded.

  This option has to be a tuple `(width, height)`, denoting the internal output shape of the the hypernet. The number of output channels is assumed to be 1, except if specified otherwise via `(width, height, channels)`.
- **`num_layers`** (*int*) – The number of transpose convolutional layers including the initial fully-connected layer.
- **`num_filters`** (*list, optional*) – List of integers of length `num_layers-1`. The number of output channels in each hidden transpose conv. layer. By default, the number of filters in the last hidden layer will be `128` and doubled in every prior layer.

  Note: The output of the first layer (which is fully-connected) is here considered to be in the shape of an image tensor.

- **`kernel_size`** (*int, tuple or list, optional*) – A single number, a tuple `(k_x, k_y)` or a list of scalars/tuples of length `num_layers-1`. Specifying the kernel size in each convolutional layer.
- **`sa_units`** (*tuple or list*) – List of integers, each representing the index of a layer in this network after which a self-attention unit should be inserted. For instance, index 0 represents the fully-connected layer. The last layer may not be chosen.

Initializes internal Module state, shared by both nn.Module and ScriptModule.

`distillation_targets`()

Targets to be distilled after training.

See docstring of abstract super method `mnets.mnet_interface.MainNetInterface.distillation_targets()`.

This network does not have any distillation targets.

Returns
`None`

`forward`(*uncond_input=None*, *cond_input=None*, *cond_id=None*, *weights=None*, *distilled_params=None*, *condition=None*, *ret_format='squeezed'*)

Compute the weights of a target network.

Parameters
(....) – See docstring of method `hnets.mlp_hnet.HMLP.forward()`.

Returns
See docstring of method `hnets.hnet_interface.HyperNetInterface.forward()`.

Return type
(list or torch.Tensor)

`get_cond_in_emb`(*cond_id*)

Get the `cond_id`-th (conditional) input embedding.

Parameters
cond_id (*int*) – Determines which input embedding should be returned (the ID has to be between `0` and `num_cond_embs-1`, where `num_cond_embs` denotes the corresponding constructor argument).

Returns
(torch.nn.Parameter)

`training: bool`

2.5 Hypernetwork-container that wraps a mixture of hypernets

The module `hnets.hnet_container` contains a hypernetwork container, i.e., a hypernetwork that produces weights by internally using a mixture of hypernetworks that implement the interface `hnets.hnet_interface.HyperNetInterface`. The container also allows the specification of shared or condition-specific weights.

Example

Assume a target network with shapes `target_shapes=[[10, 5], [5], [5], [5], [5, 5]]`, where the first 4 tensors represent the weight matrix, bias vector and batch norm scale and shift, while the last tensor is the linear output layer's weight matrix.

We consider two usecase scenarios. In the first one, the first layer weights (matrix and bias vector) are generated by a hypernetwork, while the batch norm weights should be realized via a fixed set of shared weights. The output weights shall be condition-specific:

```
from hnets import HMLP

# First-layer weights.
fl_hnet = HMLP([[10, 5], [5]], num_cond_embs=5)

def assembly_fct(list_of_hnet_tensors, uncond_tensors, cond_tensors):
    assert len(list_of_hnet_tensors) == 1
    return list_of_hnet_tensors[0] + uncond_tensors + cond_tensors

hnet = HContainer([[10, 5], [5], [5], [5], [5, 5]], assembly_fct,
                  hnets=[fl_hnet], uncond_param_shapes=[[5], [5]],
                  cond_param_shapes=[[5, 5]],
                  uncond_param_names=['bn_scale', 'bn_shift'],
                  cond_param_names=['weight'], num_cond_embs=5)
```

In the second usecase scenario, we utilize two separate hypernetworks, one as above and a second one for the condition-specific output weights. Batchnorm weights remain to be realized via a single set of shared weights.

```
from hnets import HMLP

# First-layer weights.
fl_hnet = HMLP([[10, 5], [5]], num_cond_embs=5)
# Last-layer weights.
ll_hnet = HMLP([[5, 5]], num_cond_embs=5)

def assembly_fct(list_of_hnet_tensors, uncond_tensors, cond_tensors):
    assert len(list_of_hnet_tensors) == 2
    return list_of_hnet_tensors[0] + uncond_tensors + \
        list_of_hnet_tensors[1]

hnet = HContainer([[10, 5], [5], [5], [5], [5, 5]], assembly_fct,
                  hnets=[fl_hnet, ll_hnet],
                  uncond_param_shapes=[[5], [5]],
                  uncond_param_names=['bn_scale', 'bn_shift'],
                  num_cond_embs=5)
```

class hypnettorch.hnets.hnet_container.**HContainer**(*target_shapes*, *assembly_fct*, *hnets=None*, *uncond_param_shapes=None*, *cond_param_shapes=None*, *uncond_param_names=None*, *cond_param_names=None*, *verbose=True*, *no_uncond_weights=False*, *no_cond_weights=False*, *num_cond_embs=1*)

Bases: Module, *HyperNetInterface*

Implementation of a wrapper that abstracts the use of a set of hypernetworks.

Note: Parameter tensors instantiated by this constructor are initialized via a normal distribution $\mathcal{N}(0, 0.02)$.

Parameters

- **(....)** – See constructor arguments of class hnets.mlp_hnet.HMLP.
- **assembly_fct** (*func*) – A function handle that takes the produced tensors of each internal *hypernet* (see arguments hnets, uncond_param_shapes and cond_param_shapes) and converts them into tensors with shapes target_shapes.

  The function handle must have the signature: assembly_fct(list_of_hnet_tensors, uncond_tensors, cond_tensors) . The first argument is a list of lists of tensors, the reamining two are lists of tensors. hnet_tensors contains the output of each hypernetwork in hnets. uncond_tensors contains all internally maintained unconditional weights as specified by uncond_param_shapes. cond_tensors contains the internally maintained weights corresponding to the selected condition and as specified by argument cond_param_shapes. The function is expected to return a list of tensors, each of them having a shape as specified by target_shapes.

Example

Assume `target_shapes=[[3], [3], [10, 5], [5]]` and that `hnets` is made up of two hypernetworks with output shapes `[[3]]` and `[[3], [10, 5]]`. In addition `cond_param_shapes=[[5]]`. Then the argument `hnet_tensors` will be a list of lists of tensors as follows: `[[tensor(3)], [tensor(3), tensor(10, 5)]`, `uncond_tensors` will be an empty list and `cond_tensors` will be list of tensors: `[[tensor(5)]]`.

The output of `assembly_fct` is expected to be a list of tensors as follows: `[tensor(3), tensor(3), tensor(10, 5), tensor(5)]`.

Note: This function considers one sample at a time, even if a batch of inputs is processed.

Note: It is assumed that `assembly_fct` does not further process the incoming weights. Otherwise, the attributes `mnets.mnet_interface.MainNetInterface.has_fc_out` and `mnets.mnet_interface.MainNetInterface.has_linear_out` might be invalid.

- **hnets** (*list, optional*) – List of instances of class `hnets.hnet_interface.HyperNetInterface`. All these hypernetworks are assumed to produce a part of the weights that are then assembled to a common hypernetwork output via the `assembly_fct`.
- **uncond_param_shapes** (*list, optional*) – List of lists of integers. Each entry in the list encodes the shape of an (unconditional) parameter tensor that will be added to attribute `hnets.hnet_interface.HyperNetInterface.unconditional_params` and additionally will also become an output of this hypernetwork that is passed to the `assembly_fct`.

  Hence, these parameters are independent of the hypernetwork input. Thus, they are just treated as normal weights as if they were part of the main network. This option therefore only provides the convinience of mimicking the behavior weights would elicit if they were part of the main network without needing to change the main network its implementation.
- **cond_param_shapes** (*list, optional*) – List of lists of integers. Each entry in the list encodes the shape of a (conditional) parameter tensor that will be added to attribute `hnets.hnet_interface.HyperNetInterface.conditional_params` (how often it will be added is determined by argument `num_cond_embs`). It is otherwise similar to option `uncond_param_shapes`.

  Note: If this option is specified, then argument `cond_id` of *forward()* has to be specified.

- **uncond_param_names** (*list, optional*) – If provided, it must have the same length as `uncond_param_shapes`. It will contain a list of strings that are used as values for key `name` in attribute `hnets.hnet_interface.HyperNetInterface.param_shapes_meta`.

  If not provided, shapes with more than 1 element are assigned value `weights` and all others are assigned value `bias`.
- **cond_param_names** (*list, optional*) – Same as argument `uncond_param_names` for argument `cond_param_shapes`.

Initializes internal Module state, shared by both nn.Module and ScriptModule.

`distillation_targets()`

Targets to be distilled after training.

See docstring of abstract super method `mnets.mnet_interface.MainNetInterface.distillation_targets()`.

This network does not have any distillation targets.

Returns
`None`

forward(*uncond_input=None*, *cond_input=None*, *cond_id=None*, *weights=None*, *distilled_params=None*, *condition=None*, *ret_format='squeezed'*)

Compute the weights of a target network.

Parameters

- **(....)** – See docstring of method `hnets.mlp_hnet.HMLP.forward()`. Some further information is provided below.
- **uncond_input** (*optional*) – Passed to underlying hypernetworks (see constructor argument `hnets`).
- **cond_input** (*optional*) – Passed to underlying hypernetworks (see constructor argument `hnets`).
- **cond_id** (*int or list, optional*) – Only passed to underlying hypernetworks (see constructor argument `hnets`) if `cond_input` is `None`.
- **weights** (*list or dict, optional*) – If provided as `dict` then an additional key `hnets` can be specified, which has to a list of the same length as the constructor argument `hnets` containing dictionaries as entries that will be concatenated to the extracted (hnet-specific) keys `uncond_weights` and `cond_weights`.

  For instance, for an instance of class `hnets.chunked_mlp_hnet.ChunkedHMLP` the additional key `chunk_embs` might be added.
- **condition** (*optional*) – Will be passed to the underlying hypernetworks (see constructor argument `hnets`).

Returns
See docstring of method `hnets.hnet_interface.HyperNetInterface.forward()`.

Return type
(list or torch.Tensor)

property internal_hnets

The list of internal hypernetworks provided via constructor argument `hnets`.

If `hnets` was not provided, the attribute is an empty list.

Type
list

training: `bool`

2.6 Helper functions for hypernetworks

The module `hnets.hnet_helpers` contains utilities that should simplify working with hypernetworks that implement the interface `hnets.hnet_interface.HyperNetInterface`. Those helper functions are meant to handle common manipulations (such as embedding initialization) in an abstract way that hides implementation details to the user.

hypnettorch.hnets.hnet_helpers.**get_conditional_parameters**(*hnet*, *cond_id*)

Get condition specific parameters from the hypernetwork.

Example

Class `hnets.mlp_hnet.HMLP` may only have one embedding (the conditional input embedding) per condition as conditional parameter. Thus, this function will simply return `[hnet.get_cond_in_emb(cond_id)]`.

Parameters

- **hnet** (hnets.hnet_interface.HyperNetInterface) – The hypernetwork whose conditional parameters regarding `cond_id` should be extraced.
- **cond_id** (*int*) – The condition (or its conditional ID) for which parameters should be extraced.

Returns

A list of tensors, a subset of attribute
`hnets.hnet_interface.HyperNetInterface.conditional_params`, that are specific to the condition `cond_id`. An empty list is returned if conditional parameters are not maintained internally.

Return type
(list)

hypnettorch.hnets.hnet_helpers.**init_chunk_embeddings**(*hnet*, *normal_mean=0.0*, *normal_std=1.0*, *init_fct=None*)

Initialize chunk embeddings.

This function only applies to hypernetworks that make use of chunking, such as `hnets.chunked_mlp_hnet.ChunkedHMLP`. All other hypernetwork types will be unaffected by this function.

This function handles the initialization of embeddings very similar to function *init_conditional_embeddings()*, except that the function handle `init_fct` has a slightly different signature. It receives two positional arguments, the chunk embedding and the chunk embedding ID as well as one optional argument `cond_id`, the conditional ID (in case of conditional chunk embeddings).

```
init_fct = lambda cemb, cid, cond_id=None : nn.init.constant_(cemb, 0)
```

Note: Class `hnets.structured_mlp_hnet.StructuredHMLP` has multiple sets of chunk tensors as specified by attribute `hnets.structured_mlp_hnet.StructuredHMLP.chunk_emb_shapes`. As a simplifying design choice, the tensors passed to `init_fct` will not be single embeddings (i.e., vectors), but tensors of embeddings according to the shapes in attribute `hnets.structured_mlp_hnet.StructuredHMLP.chunk_emb_shapes`.

Parameters
(....) – See docstring of function *init_conditional_embeddings()*.

hypnettorch.hnets.hnet_helpers.**init_conditional_embeddings**(*hnet*, *normal_mean=0.0*, *normal_std=1.0*, *init_fct=None*)

Initialize internally maintained conditional input embeddings.

This function initializes conditional embeddings if the hypernetwork has any and they are internally maintained. For instance, the conditional embeddings of an `HMLP` instance are those returned by the method `hnets.mlp_hnet.HMLP.get_cond_in_emb()`.

By default, those embedding will follow a normal distribution. However, one may pass a custom init function `init_fct` that receives the embedding and its corresponding conditional ID as input (as is expected to modify the embedding in-place):

```
init_fct(cond_emb, cond_id)
```

Hypernetworks that don't make use of internally maintained conditional input embeddings will not be affected by this function.

Note: Chunk embeddings may also be conditional parameters, but are not considered conditional input embeddings here. Conditional chunk embeddings can be initialized using function *init_chunk_embeddings()*.

Parameters

- **hnet** (hnets.hnet_interface.HyperNetInterface) – The hypernetwork whose conditional embeddings should be initialized.
- **normal_mean** (*float*) – The mean of the normal distribution with which embeddings should be initialized.
- **normal_std** (*float*) – The std of the normal distribution with which embeddings should be initialized.
- **init_fct** (*func, optional*) – A function handle that receives a conditional embedding and its ID as input and initializes the embedding in-place. If provided, arguments `normal_mean` and `normal_std` will be ignored.

2.7 Hypernetwork-wrapper for input-preprocessing and output-postprocessing

The module `hnets.hnet_perturbation_wrapper` implements a wrapper for hypernetworks that implement the interface `hnets.hnet_interface.HyperNetInterface`. By default, the wrapper is meant for perturbing hypernetwork outputs, such that an implicit distribution (realized via a hypernetwork) with low-dimensional support can be inflated to have support in the full weight space.

However, the wrapper allows in general to pass function handles that preprocess inputs and/or postprocess hypernetwork outputs.

class hypnettorch.hnets.hnet_perturbation_wrapper.**HPerturbWrapper**(*hnet*, *hnet_uncond_in_size=None*, *sigma_noise=0.02*, *input_handler=None*, *output_handler=None*, *verbose=True*)

Bases: Module, *HyperNetInterface*

Hypernetwork wrapper for output perturbation.

This wrapper is meant as a helper for hypernetworks that represent implicit distributions, i.e., distributions that transform a simple base distribution $p_Z(z)$ into a complex target distributions

$$w \sim q_\theta(W) \Leftrightarrow w = h_\theta(z) \quad , \quad z \sim p_Z(Z)$$

However, the wrapper is more versatile and can also become handy in a variety of other use cases. Yet, in the following we concentrate on implicit distributions and their practical challenges. One main challenge is typically that the density $q_\theta(W)$ is only defined on a lower-dimensional manifold of the weight space. This is often an undesirable property (e.g., such implicit distributions are often not amenable for optimization with standard divergence measures, such as the KL).

A simple way to overcome this issue is to add noise perturbations to the output of the hypernetwork, such that the perturbations itself origin from a full-support distribution. By default, this hypernetwork wrapper adjusts the sampling procedure above in the following way

$$w \sim \tilde{q}_\theta(W) \Leftrightarrow w = h_\theta(z_{:n}) + \sigma^2_{\text{noise}} z \equiv \tilde{h}_\theta(z) \quad , \quad z \sim p_Z(Z) \tag{2.1}$$

where now $\dim(\mathcal{W}) = \dim(\mathcal{Z})$, σ_{noise} is a hyperparameter that controls the perturbation strength, and $z_{:n}$ are the n first entries of the vector z.

By default, the unconditional input size of this hypernetwork will be of size `hnet.num_outputs` (if `input_handler` is not provided) and the output size will be of the same size.

Parameters

- **hnet** (hnets.hnet_interface.HyperNetInterface) – The hypernetwork around which this wrapper should be wrapped.
- **hnet_uncond_in_size** (*int*) – This argument refers to n from Eq. (2.1). If `input_handler` is provided, this argument will be ignored.
- **sigma_noise** (*float*) – The perturbation strength σ_{noise} from Eq. (2.1). If `output_handler` is provided, this argument will be ignored.
- **input_handler** (*func, optional*) – A function handler to process the inputs to the `hnets.hnet_interface.HyperNetInterface.forward()` method of `hnet`. The function handler should have the following signature

```
uncond_input_int, cond_input_int, cond_id_int = input_handler( \
    uncond_input=None, cond_input=None, cond_id=None)
```

  The returned values will be passed to *internal_hnet*.

  ### Example

  For instance, to reproduce the behavior depicted in Eq. (2.1) one could provide the following handler

```
def input_handler(uncond_input=None, cond_input=None,
                  cond_id=None):
    assert uncond_input is not None
    n = 5
    return uncond_input[:, :n], cond_input, cond_id
```

- **output_handler** (*func, optional*) – A function handler to postprocess the outputs of the internal hypernetwork *internal_hnet*.

  A function handler with the following signature is expected.

```
hnet_out = output_handler(hnet_out_int, uncond_input=None,
                          cond_input=None, cond_id=None)
```

where `hnet_out_int` is the output of the internal hypernetwork *internal_hnet* and the remaining arguments are the original arguments passed to method *forward()*. `hnet_out_int` will always have the format `ret_format='flattened'` and is also expected to return this format.

Example

Deviating from Eq. (2.1), let's say we want to implement the following sampling behavior

$$w \sim \hat{q}_\theta(W) \Leftrightarrow w = h_\theta(z) + \epsilon_w \quad , \quad z \sim p_Z(Z) \text{ and } \epsilon_w \sim p_{\text{noise}}(W)$$

In this case the unconditional input `uncond_input` to the *forward()* method is expected to have size $\dim(\mathcal{Z}) + \dim(\mathcal{W})$.

```
def input_handler(uncond_input=None, cond_input=None,
                  cond_id=None):
    assert uncond_input is not None
    return uncond_input[:, :dim_z], cond_input, cond_id
```

```
def output_handler(hnet_out_int, uncond_input=None,
                   cond_input=None, cond_id=None):
    assert uncond_input is not None
    return hnet_out_int + uncond_input[:, dim_z:]
```

- **verbose** (*bool*) – Whether network information should be printed during network creation.

Initializes internal Module state, shared by both nn.Module and ScriptModule.

distillation_targets()

Targets to be distilled after training.

See docstring of abstract super method `mnets.mnet_interface.MainNetInterface.distillation_targets()`.

Returns
Simply returns the `distillation_targets` of the internal hypernet `internal_hnet`.

forward(*uncond_input=None*, *cond_input=None*, *cond_id=None*, *weights=None*, *distilled_params=None*, *condition=None*, *ret_format='squeezed'*)

Compute the weights of a target network.

Parameters
(....) – See docstring of method `hnets.hnet_interface.HyperNetInterface.forward()`.

Returns
See docstring of method `hnets.hnet_interface.HyperNetInterface.forward()`.

Return type
(list or torch.Tensor)

property internal_hnet

The underlying hypernetwork that was passed via constructor argument `hnet`.

Type
hnets.hnet_interface.HyperNetInterface

`training: bool`

2.8 MLP - Hypernetwork

The module `hnets.mlp_hnet` contains a fully-connected hypernetwork (also termed *full hypernet*).

This type of hypernetwork represents one of the most simplistic architectural choices to realize a weight generator. An embedding input, which may consists of conditional and unconditional parts (for instance, in the case of task-conditioned hypernetwork the conditional input will be a task embedding) is mapped via a series of fully-connected layers onto a final hidden representation. Then a linear fully-connected output layer per is used to produce the target weights, output tensors with shapes specified via the target shapes (see `hnets.hnet_interface.HyperNetInterface.target_shapes`).

If no hidden layers are used, then this resembles a simplistic linear hypernetwork, where the input embeddings are linearly mapped onto target weights.

`class hypnettorch.hnets.mlp_hnet.HMLP(`*target_shapes*, *uncond_in_size=0*, *cond_in_size=8*, *layers=(100, 100)*, *verbose=True*, *activation_fn=ReLU()*, *use_bias=True*, *no_uncond_weights=False*, *no_cond_weights=False*, *num_cond_embs=1*, *dropout_rate=-1*, *use_spectral_norm=False*, *use_batch_norm=False*`)`

Bases: `Module`, *`HyperNetInterface`*

Implementation of a *full hypernet*.

The network will consist of several hidden layers and a final linear output layer that produces all weight matrices/bias-vectors the network has to produce.

The network allows to maintain a set of embeddings internally that can be used as conditional input.

Parameters

- **`target_shapes`** (*`list`*) – List of lists of intergers, i.e., a list of tensor shapes. Those will be the shapes of the output weights produced by the hypernetwork. For each entry in this list, a separate output layer will be instantiated.
- **`uncond_in_size`** (*`int`*) – The size of unconditional inputs (for instance, noise).
- **`cond_in_size`** (*`int`*) – The size of conditional input embeddings.

  Note, if `no_cond_weights` is `False`, those embeddings will be maintained internally.
- **`layers`** (*`list or tuple`*) – List of integers denoteing the sizes of each hidden layer. If empty, no hidden layers will be produced.
- **`verbose`** (*`bool`*) – Whether network information should be printed during network creation.
- **`activation_fn`** (*`func`*) – The activation function to be used for hidden activations. For instance, an instance of class `torch.nn.ReLU`.
- **`use_bias`** (*`bool`*) – Whether the fully-connected layers that make up this network should have bias vectors.
- **`no_uncond_weights`** (*`bool`*) – If `True`, unconditional weights are not maintained internally and instead expected to be produced externally and passed to the *`forward()`*.

- **no_cond_weights** (*bool*) – If `True`, conditional embeddings are assumed to be maintained externally. Otherwise, option `num_cond_embs` has to be properly set, which will determine the number of embeddings that are internally maintained.
- **num_cond_embs** (*int*) – Number of conditional embeddings to be internally maintained. Only used if option `no_cond_weights` is `False`.

  Note: Embeddings will be initialized with a normal distribution using zero mean and unit variance.

- **dropout_rate** (*float*) – If `-1`, no dropout will be applied. Otherwise a number between 0 and 1 is expected, denoting the dropout rate of hidden layers.
- **use_spectral_norm** (*bool*) – Use spectral normalization for training.
- **use_batch_norm** (*bool*) – Whether batch normalization should be used. Will be applied before the activation function in all hidden layers.

  Note: Batch norm only makes sense if the hypernetwork is envoked with batch sizes greater than 1 during training.

Initializes internal Module state, shared by both nn.Module and ScriptModule.

apply_hyperfan_init(*method='in'*, *use_xavier=False*, *uncond_var=1.0*, *cond_var=1.0*, *mnet=None*, *w_val=None*, *w_var=None*, *b_val=None*, *b_var=None*)

Initialize the network using *hyperfan init*.

Hyperfan initialization was developed in the following paper for this kind of hypernetwork

> "Principled Weight Initialization for Hypernetworks" https://openreview.net/forum?id=H1lma24tPB

The initialization is based on the following idea: When the main network would be initialized using Xavier or Kaiming init, then variance of activations (fan-in) or gradients (fan-out) would be preserved by using a proper variance for the initial weight distribution (assuming certain assumptions hold at initialization, which are different for Xavier and Kaiming).

When using this kind of initializations in the hypernetwork, then the variance of the initial main net weight distribution would simply equal the variance of the input embeddings (which can lead to exploding activations, e.g., for fan-in inits).

The above mentioned paper proposes a quick fix for the type of hypernet that resembles the simple MLP hnet implemented in this class, i.e., which have a separate output head per weight tensor in the main network.

Assuming that input embeddings are initialized with a certain variance (e.g., 1) and we use Xavier or Kaiming init for the hypernet, then the variance of the last hidden activation will also be 1.

Then, we can modify the variance of the weights of each output head in the hypernet to obtain the same variance per main net weight tensor that we would typically obtain when applying Xavier or Kaiming to the main network directly.

Note: If `mnet` is not provided or the corresponding attribute `mnets.mnet_interface.MainNetInterface.param_shapes_meta` is not implemented, then this method assumes that 1D target tensors (cf. constructor argument `target_shapes`) represent bias vectors in the main network.

Note: To compute the hyperfan-out initialization of bias vectors, we need access to the fan-in of the layer, which we can only compute based on the corresponding weight tensor in the same layer. This is only possible if `mnet` is provided. Otherwise, the following heuristic is applied. We assume that the shape directly preceding a bias shape in the constructor argument `target_shapes` is the corresponding weight tensor.

Note: All hypernet inputs are assumed to be zero-mean random variables.

Variance of the hypernet input

In general, the input to the hypernetwork can be a concatenation of multiple embeddings (see description of arguments `uncond_var` and `cond_var`).

Let's denote the complete hypernetwork input by $\mathbf{x} \in \mathbb{R}^n$, which consists of a conditional embedding $\mathbf{e} \in \mathbb{R}^{n_e}$ and an unconditional input $\mathbf{c} \in \mathbb{R}^{n_c}$, i.e.,

$$\mathbf{x} = \begin{bmatrix} \mathbf{e} \\ \mathbf{c} \end{bmatrix}$$

We simply define the variance of an input $\text{Var}(x_j)$ as the weighted average of the individual variances, i.e.,

$$\text{Var}(x_j) \equiv \frac{n_e}{n_e + n_c} \text{Var}(e) + \frac{n_c}{n_e + n_c} \text{Var}(c)$$

To see that this is correct, consider a linear layer $\mathbf{y} = W\mathbf{x}$ or

$$\begin{aligned} y_i &= \sum_j w_{ij} x_j \\ &= \sum_{j=1}^{n_e} w_{ij} e_j + \sum_{j=n_e+1}^{n_e+n_c} w_{ij} c_{j-n_e} \end{aligned}$$

Hence, we can compute the variance of y_i as follows (assuming the typical Xavier assumptions):

$$\begin{aligned} \text{Var}(y) &= n_e \text{Var}(w) \text{Var}(e) + n_c \text{Var}(w) \text{Var}(c) \\ &= \frac{n_e}{n_e + n_c} \text{Var}(e) + \frac{n_c}{n_e + n_c} \text{Var}(c) \end{aligned}$$

Note, that Xavier would have initialized W using $\text{Var}(w) = \frac{1}{n} = \frac{1}{n_e + n_c}$.

Parameters

- **method** (*str*) – The type of initialization that should be applied. Possible options are:
  - `'in'`: Use *Hyperfan-in*.
  - `'out'`: Use *Hyperfan-out*.
  - `'harmonic'`: Use the harmonic mean of the *Hyperfan-in* and *Hyperfan-out* init.
- **use_xavier** (*bool*) – Whether Kaiming (`False`) or Xavier (`True`) init should be used.
- **uncond_var** (*float*) – The variance of unconditional embeddings. This value is only taken into consideration if `uncond_in_size > 0` (cf. constructor arguments).
- **cond_var** (*float*) – The initial variance of conditional embeddings. This value is only taken into consideration if `cond_in_size > 0` (cf. constructor arguments).

- **mnet** (mnets.mnet_interface.MainNetInterface, *optional*) – If applicable, the user should provide the main (or target) network, whose weights are generated by this hypernetwork. The `mnet` instance is used to extract valuable information that improve the initialization result. If provided, it is assumed that `target_shapes` (cf. constructor arguments) corresponds either to `mnets.mnet_interface.MainNetInterface.param_shapes` or `mnets.mnet_interface.MainNetInterface.hyper_shapes_learned`.

- **w_val** (*list or dict, optional*) – The mean of the distribution with which output head weight matrices are initialized. Note, each weight tensor prescribed by `hnets.hnet_interface.HyperNetInterface.target_shapes` is produced via an independent linear output head.

  One may either specify a list of numbers having the same length as `hnets.hnet_interface.HyperNetInterface.target_shapes` or specify a dictionary which may have as keys the tensor names occurring in `mnets.mnet_interface.MainNetInterface.param_shapes_meta` and the corresponding mean value for the weight matrices of all output heads producing this type of tensor. If a list is provided, entries may be `None` and if a dictionary is provided, not all types of parameter tensors need to be specified. For tensors, for which no value is specified, the default value will be used. The default values for tensor types `'weight'` and `'bias'` are calculated based on the proposed hyperfan-initialization. For other tensor types the actual hypernet outputs should be drawn from the following distributions

  - `'bn_scale'`: $w \sim \delta(w - 1)$
  - `'bn_shift'`: $w \sim \delta(w)$
  - `'cm_scale'`: $w \sim \delta(w - 1)$
  - `'cm_shift'`: $w \sim \delta(w)$
  - `'embedding'`: $w \sim \mathcal{N}(0, 1)$

  Which would correspond to the following passed arguments

```
w_val = {
    'bn_scale': 0,
    'bn_shift': 0,
    'cm_scale': 0,
    'cm_shift': 0,
    'embedding': 0
}
w_var = {
    'bn_scale': 0,
    'bn_shift': 0,
    'cm_scale': 0,
    'cm_shift': 0,
    'embedding': 0
}
b_val = {
    'bn_scale': 1,
    'bn_shift': 0,
    'cm_scale': 1,
    'cm_shift': 0,
    'embedding': 0
}
b_var = {
```

(continues on next page)

(continued from previous page)

```
    'bn_scale': 0,
    'bn_shift': 0,
    'cm_scale': 0,
    'cm_shift': 0,
    'embedding': 1
}
```

- **w_var** (*list or dict, optional*) – The variance of the distribution with which output head weight matrices are initialized. Variance values of zero means that weights are set to a constant defined by `w_val`. See description of argument `w_val` for more details.
- **b_val** (*list or dict, optional*) – The mean of the distribution with which output head bias vectors are initialized. See description of argument `w_val` for more details.
- **b_var** (*list or dict, optional*) – The variance of the distribution with which output head bias vectors are initialized. See description of argument `w_val` for more details.

distillation_targets()

Targets to be distilled after training.

See docstring of abstract super method `mnets.mnet_interface.MainNetInterface.distillation_targets()`.

This network does not have any distillation targets.

Returns
None

forward(*uncond_input=None*, *cond_input=None*, *cond_id=None*, *weights=None*, *distilled_params=None*, *condition=None*, *ret_format='squeezed'*)

Compute the weights of a target network.

Parameters

- **(....)** – See docstring of method `hnets.hnet_interface.HyperNetInterface.forward()`.
- **condition** (*int, optional*) – This argument will be passed as argument `stats_id` to the method `utils.batchnorm_layer.BatchNormLayer.forward()` if batch normalization is used.

Returns
See docstring of method `hnets.hnet_interface.HyperNetInterface.forward()`.

Return type
(list or torch.Tensor)

get_cond_in_emb(*cond_id*)

Get the `cond_id`-th (conditional) input embedding.

Parameters
cond_id (*int*) – Determines which input embedding should be returned (the ID has to be between `0` and `num_cond_embs-1`, where `num_cond_embs` denotes the corresponding constructor argument).

Returns
(torch.nn.Parameter)

training: bool

2.9 Example Instantiations of a Structured Chunked MLP - Hypernetwork

The module `hnets.structured_hmlp_examples` provides helpers for example instantiations of `hnets.structured_mlp_hnet.StructuredHMLP`.

Functions in this module typically take a given main network and produce the constructor arguments `chunk_shapes`, `num_per_chunk` and `assembly_fct` of class `hnets.structured_mlp_hnet.StructuredHMLP`.

Note: These examples should be used with care. They are meant as inspiration and might not cover all possible usecases.

hypnettorch.hnets.structured_hmlp_examples.resnet_chunking(net)	Design a structured chunking for a ResNet.
hypnettorch.hnets.structured_hmlp_examples.wrn_chunking(net)	Design a structured chunking for a Wide-ResNet (WRN).

hypnettorch.hnets.structured_hmlp_examples.**resnet_chunking**(*net*, *gcd_chunking=False*)

Design a structured chunking for a ResNet.

A resnet as implemented in class `mnets.resnet.ResNet` consists roughly of 5 parts:

- An input convolutional layer with weight shape `[C_1, C_in, 3, 3]`
- 3 blocks of `2*n` convolutional layers each where the first layer has shape `[C_i, C_j, 3, 3]` with $i \in \{2, 3, 4\}$ and $j \equiv i - 1$ and the remaining `2*n-1` layers have a weight shape of `[C_i, C_i, 3, 3]`.
- A final fully connected layer of shape `[n_classes, n_hidden]`.

Each layer may additionally have a bias vector and (if batch normalization is used) a scale and shift vector.

For instance, if a resnet with biases and batchnorm is used and the first layer will be produced as one structured chunk, then the first chunk shape (see return value `chunk_shapes`) will be: `[[C_1, C_in, 3, 3], [C_1], [C_1], [C_1]]`.

This function will chunk layer wise (i.e., a chunk always comprises up to 4 elements: weights tensor, bias vector, batchnorm scale and shift). By default, layers with the same shape are grouped together. Hence, the standard return value contains 8 chunk shapes (input layer, first layer of each block, remaining layers of each block (which all have the same shape) and the fully-connected output layer). Therefore, the return value `num_per_chunk` would be as follows: `[1, 1, 2*n-1, 1, 2*n-1, 1, 2*n-1, 1]`.

Parameters

- **net** (*mnets.resnet.ResNet*) – The network for which the structured chunking should be devised.
- **gcd_chunking** (*bool*) – If `True`, the layers within the 3 resnet blocks will be produced by 4 chunks. Therefore, the greatest common divisor (gcd) of the feature sizes `C_1`, `C_2`, `C_3`, `C_4` is computed and the 6 middle `chunk_shapes` produced by default are replaced by 4 chunk shapes `[[C_gcd, C_i, 3, 3], [C_gcd]]` (assuming no batchnorm is used). Note, the first and last entry of `chunk_shapes` will remain unchanged by this option.

  Hence, `len(num_per_chunk) = 6` in this case.

Returns

Tuple containing the following arguments that can be passed to the constructor of class `hnets.structured_mlp_hnet.StructuredHMLP`.

- **chunk_shapes** (list)
- **num_per_chunk** (list)
- **assembly_fct** (func)

Return type
(tuple)

hypnettorch.hnets.structured_hmlp_examples.**wrn_chunking**(*net*, *ignore_bn_weights=True*, *ignore_out_weights=True*, *gcd_chunking=False*)

Design a structured chunking for a Wide-ResNet (WRN).

This function is in principle similar to function *resnet_chunking()*, but with the goal to provide a chunking scheme that is identical to the one proposed in (accessed August 18th, 2020):

> Sacramento et al., "Economical ensembles with hypernetworks", 2020 https://arxiv.org/abs/2007.12927

Therefore, a WRN as implemented in class `mnets.wide_resnet.WRN` is required. For instance, a *WRN-28-10-B(3,3)* can be instantiated as follows, using batchnorm but no biases in all convolutional layers:

```
wrn = WRN(in_shape=(32, 32, 3), num_classes=10, n=4, k=10,
          num_feature_maps=(16, 16, 32, 64), use_bias=False,
          use_fc_bias=True, no_weights=False, use_batch_norm=True)
```

We denote channel sizes by `[C_in, C_1, C_2, C_3, C_4]`, where `C_in` is the number of input channels and the remaining `C_1, C_2, C_3, C_4` denote the channel size per convolutional group. The widening factor is denoted by `k`.

In general, there will be up to 11 *layer groups*, which will be realized by separate hypernetworks (cmp table S1 in Sacramento et al.):

- `0`: Input layer weights. If the network's convolutional layers have biases and batchnorm layers while `ignore_bn_weights=False`, then this hypernet will produce weights of shape `[[C_1, C_in, 3, 3], [C_1], [C_1], [C_1]]`. However, without convolutional bias terms and with `ignore_bn_weights=True`, the hypernet will only produce weights of shape `[[C_1, C_in, 3, 3]]`. This specification applies to all layer groups generating convolutional layers.
- `1`: This layer group will generate the weights of the first convolutional layer in the first convolutional group, e.g., `[[k*C_2, C_1, 3, 3]]`. Let's define `r = max(k*C_2/C_1, C_1/k*C_2)`. If `r=1` or `r=2` or `gcd_chunking=True`, then this group is merged with layer group `2`.
- `2`: The remaining convolutional layer of the first convolutional group. If `r=1`, `r=2` or `gcd_chunking=True`, then all convolutional layers of the first group are generated. However, if biases or batch norm weights have to be generated, then this form of chunking leads to redundancy. Imagine bias terms are used and that the first layer in this convolutional group has weights `[[160, 16, 3, 3], [160]]`, while the remaining layers have shape `[[160, 160, 3, 3], [160]]`. If that's the case, the hypernetwork output will be of shape `[[160, 16, 3, 3], [160]]`, meaning that 10 chunks have to be produced for each except the first layer. However, this means that per convolutional layer 10 bias vectors are generated, while only one is needed and therefore the other 9 will go to waste.
- `3`: Same as `1` for the first layer in the second convolutional group.
- `4` (labelled as `3` in the paper): Same as `2` for all convolutional layers (potentially excluding the first) in the second convolutional group.

- 5: Same as 1 for the first layer in the third convolutional group.
- 6 (labelled as 4 in the paper): Same as 2 for all convolutional layers (potentially excluding the first) in the third convolutional group.
- 7 (labelled as 5 in the paper): If existing, this hypernetwork produces the 1x1 convolutional layer realizing the residual connection connecting the first and second residual block in the first convolutional group.
- 8 (labelled as 6 in the paper): Same as 7 but for the first residual connection in the second convolutional group.
- 9 (labelled as 7 in the paper): Same as 7 but for the first residual connection in the third convolutional group.
- 10: This hypernetwork will produce the weights of the fully connected output layer, if ignore_out_weights=False.

Thus, the WRN weights would maximally be produced by 11 different sub- hypernetworks.

Note: There is currently an implementation mismatch, such that the implementation provided here does not 100% mimic the architecture described in Sacramento et al..

To be specific, given the wrn generated above, the hypernetwork output for layer group 2 will be of shape [160, 160, 3, 3], while the paper expects a vertical chunking with a hypernet output of shape [160, 80, 3, 3].

Parameters

- **net** (mnets.wide_resnet.WRN) – The network for which the structured chunking should be devised.
- **ignore_bn_weights** (*bool*) – If True, even if the given net has batchnorm weights, they will be ignored by this function.
- **ignore_out_weights** (*bool*) – If True, output weights (layer group 10) will be ignored by this function.
- **gcd_chunking** (*bool*) – If True, layer groups 1, 3 and 5 are ignored. Instead, the greatest common divisor (gcd) of input and output feature size in a convolutional group is computed and weight tensors within a convolutional group (i.e., layer groups 2, 4 and 6) are chunked according to this value. However, note that this will cause the generation of unused bias and batchnorm weights if existing (cp. description of layer group 2).

Returns

Tuple containing the following arguments that can be passed to the constructor of class hnets.structured_mlp_hnet.StructuredHMLP.

- **chunk_shapes** (list)
- **num_per_chunk** (list)
- **assembly_fct** (func)

Return type
(tuple)

2.10 Structured Chunked MLP - Hypernetwork

The module `hnets.structured_mlp_hnet` contains a *Structured Chunked Hypernetwork*, i.e., a hypernetwork that is aware of the target network architecture and choses a smart way of chunking.

In contrast to the *Chunked Hypernetwork* `hnets.chunked_mlp_hnet.ChunkedHMLP`, which just flattens the `target_shapes` and splits them into equally sized chunks (ignoring the underlying network structure in terms of layers or type of weight (bias, kernel, ...)), the *StructuredHMLP* aims to preserve this structure when chunking the target weights.

Example

Assume `target_shapes = [[3], [3], [10, 5], [10], [20, 5], [20]]`.

There are now many ways to split those weights into chunks. In the simplest case, we consider only one chunk and produce all weights at once with a *Full Hypernetwork* `hnets.mlp_hnet.HMLP`.

Another simple scenario would be to realize that all shapes except the first two are different. So, we create a total of 5 internal hypernetworks for those 6 weight tensors, where the first internal hypernetwork would produce weights of shape `[3]` upon receiving an external input plus an internal chunk embedding. See below for an example instantiation:

```
def assembly_fct(list_of_chunks):
    assert len(list_of_chunks) == 4
    ret = []
    for chunk in list_of_chunks:
        ret.extend(chunk)
    return ret

hnet = StructuredHMLP([[3], [3], [10, 5], [10], [20, 5], [20]],
    [[[3]], [[10, 5], [10]], [[20, 5], [20]]], [2, 1, 1], 8,
    {'layers': [10,10]}, assembly_fct, cond_chunk_embs=True,
    uncond_in_size=0, cond_in_size=0, verbose=True,
    no_uncond_weights=False, no_cond_weights=False, num_cond_embs=1)
```

A smarter way of chunking would be to realize that the last two shapes are just twice the middle two shapes. Hence, we could instantiate two internal hypernetworks. The first one would be used to produce tensors of shape `[3]` and therefore require 2 chunk embeddings. The second internal hypernetwork would be used to create tensors of shape `[10, 5]`, `[10]`, requiring 3 chunk embeddings (the last two chunks together make up the last two target tensors of shape `[20, 5]`, `[20]`).

```
def assembly_fct(list_of_chunks):
    assert len(list_of_chunks) == 5
    ret = [*list_of_chunks[0], *list_of_chunks[1], *list_of_chunks[2]]
    for t, tensor in enumerate(list_of_chunks[3]):
        ret.append(torch.cat([tensor, list_of_chunks[4][t]], dim=0))
    return ret

hnet = StructuredHMLP([[3], [3], [10, 5], [10], [20, 5], [20]],
    [[[3]], [[10, 5], [10]]], [2, 3], 8,
    {'layers': [10,10]}, assembly_fct, cond_chunk_embs=True,
    uncond_in_size=0, cond_in_size=0, verbose=True,
    no_uncond_weights=False, no_cond_weights=False, num_cond_embs=1)
```

Example

This hypernetwork can also be used to realize soft-sharing via templates as proposed in Savarese et al.

Assume a target network with 3 layers of identical weight shapes `target_shapes=[s, s, s]`, where `s` denotes a weight shape.

If we want to create these 3 weight tensors via a linear combination of two templates, we could create an instance of *StructuredHMLP* as follows:

```
def assembly_fct(list_of_chunks):
    assert len(list_of_chunks) == 3
    return [list_of_chunks[0][0], list_of_chunks[1][0],
            list_of_chunks[2][0]]

hnet = StructuredHMLP([s, s, s], [[s]], [3], 2,
    {'layers': [], 'use_bias': False}, assembly_fct
    cond_chunk_embs=True, uncond_in_size=0, cond_in_size=0,
    verbose=True, no_uncond_weights=False, no_cond_weights=False,
    num_cond_embs=1)
```

There will be one underlying linear hypernetwork, that expects a 2-dimensional embedding input. The computation of the linear hypernetwork can be seen as $t_i = We_i$. Where t_i is a tensor of shape `s` containing the weights of the i-th chunk (with chunk embedding e_i).

The 2 templates are encoded in the hypernetwork weights W, whereas the chunk embedding represents the coefficients of the linear combination.

class hypnettorch.hnets.structured_mlp_hnet.**StructuredHMLP**(*target_shapes*, *chunk_shapes*, *num_per_chunk*, *chunk_emb_sizes*, *hmlp_kwargs*, *assembly_fct*, *cond_chunk_embs=False*, *uncond_in_size=0*, *cond_in_size=8*, *verbose=True*, *no_uncond_weights=False*, *no_cond_weights=False*, *num_cond_embs=1*)

Bases: Module, *HyperNetInterface*

Implementation of a *structured chunked fully-connected hypernet*.

This network builds a series of full hypernetworks internally (hidden from the user). There will be one internal hypernetwork for each element of `chunk_shapes`. Those internal hypernetworks can produce an arbitrary amount of chunks (as defined by `num_per_chunk`). All those chunks are finally assembled by function `assembly_fct` to produce tensors according to `target_shapes`.

Note: It is possible to set `uncond_in_size` and `cond_in_size` to zero if `cond_chunk_embs` is `True` and there are no zeroes in argument `chunk_emb_sizes`.

Parameters

- **(....)** – See constructor arguments of class `hnets.mlp_hnet.HMLP`.
- **chunk_shapes** (*list*) – List of lists of lists of integers. Each chunk will be produced by its own internal hypernetwork (instance of class `hnets.mlp_hnet.HMLP`). Hence, this list can be seen as a list of `target_shapes`, passed to the underlying internal hypernets.

- **num_per_chunk** (*list*) – List of the same length as `chunk_shapes`, that determines how often each of these chunks has to be produced.
- **chunk_emb_sizes** (*list or int*) – List with the same length as `chunk_shapes` or single integer that will be expanded to this length. Determines the chunk embedding size per internal hypernetwork.

  Note: Embeddings will be initialized with a normal distribution using zero mean and unit variance.

  Note: If the corresponding entry in `num_per_chunk` is `1`, then an embedding size might be `0`, which means there won't be chunk embeddings for the corresponding internal hypernetwork.

- **hmlp_kwargs** (*list or dict*) – List of dictionaries or a single dictionary that will be expanded to such a list. Those dictionaries may contain keyword arguments for each instance of class `hnets.mlp_hnet.HMLP` that will be generated.

  The following keys are **not permitted** in these dictionaries: - `uncond_in_size` - `cond_in_size` - `no_uncond_weights` - `no_cond_weights` - `num_cond_embs` Those arguments will be determined by the corresponding keyword arguments of this class!
- **assembly_fct** (*func*) – A function handle that takes the produced chunks and converts them into tensors with shapes `target_shapes`.

  The function handle must have the signature: `assembly_fct(list_of_chunks)`. The argument `list_of_chunks` is a list of lists of tensors. The function is expected to return a list of tensors, each of them having a shape as specified by `target_shapes`.

  #### Example

  Assume `chunk_shapes=[[[3]], [[10, 5], [5]]]` and `num_per_chunk=[2, 1]`. Then the argument `list_of_chunks` will be a list of lists of tensors as follows: `[[tensor(3)], [tensor(3)], [tensor(10, 5), tensor(5)]]`.

  If `target_shapes=[[3], [3], [10, 5], [5]]`, then the output of `assembly_fct` is expected to be a list of tensors as follows: `[tensor(3), tensor(3), tensor(10, 5), tensor(5)]`.

  Note: This function considers one sample at a time, even if a batch of inputs is processed.

  Note: It is assumed that `assembly_fct` does not further process the incoming weights. Otherwise, the attributes `mnets.mnet_interface.MainNetInterface.has_fc_out` and `mnets.mnet_interface.MainNetInterface.has_linear_out` might be invalid.

- **cond_chunk_embs** (*bool*) – See documentation of class `hnets.chunked_mlp_hnet.ChunkedHMLP`

Initializes internal Module state, shared by both nn.Module and ScriptModule.

property chunk_emb_shapes

List of lists of integers. The list contains the shape of the chunk embeddings required per forward sweep.

Note: Some internal hypernets might not need chunk embeddings if the corresponding entry in `chunk_emb_sizes` is zero.

Type
list

property cond_chunk_embs

Whether chunk embeddings are unconditional (`False`) or conditional (`True`) parameters. See constructor argument `cond_chunk_embs`.

Type
bool

distillation_targets()

Targets to be distilled after training.

See docstring of abstract super method `mnets.mnet_interface.MainNetInterface.distillation_targets()`.

This network does not have any distillation targets.

Returns
`None`

forward(*uncond_input=None*, *cond_input=None*, *cond_id=None*, *weights=None*, *distilled_params=None*, *condition=None*, *ret_format='squeezed'*)

Compute the weights of a target network.

Parameters

- **(....)** – See docstring of method `hnets.mlp_hnet.HMLP.forward()`.
- **weights** (*list or dict, optional*) – If provided as `dict` and chunk embeddings are considered conditional (see constructor argument `cond_chunk_embs`), then the additional key `chunk_embs` can be used to pass a batch of chunk embeddings. This option is mutually exclusive with the option of passing `cond_id`. Note, if conditional inputs via `cond_input` are expected, then the batch sizes must agree.

  A batch of chunk embeddings is expected to be a list of tensors of shape `[B, *ce_shape]`, where `B` denotes the batch size and `ce_shape` is a shape from list *chunk_emb_shapes*.

Returns
See docstring of method `hnets.hnet_interface.HyperNetInterface.forward()`.

Return type
(list or torch.Tensor)

get_chunk_embs(*cond_id=None*)

Get the chunk embeddings.

Parameters
cond_id (*int*) – Is mandatory if constructor argument `cond_chunk_embs` was set. Determines the set of chunk embeddings to be considered.

Returns
A list of tensors with shapes prescribed by *chunk_emb_shapes*.

Return type
(list)

`get_cond_in_emb`(*cond_id*)

Get the `cond_id`-th (conditional) input embedding.

Parameters
(....) – See docstring of method `hnets.mlp_hnet.HMLP.get_cond_in_emb()`.

Returns
(torch.nn.Parameter)

`property internal_hnets`

The list of internal hypernetworks (instances of class `hnets.mlp_hnet.HMLP`) which are created to produce the individual chunks according to constructor argument `chunk_shapes`.

Type
list

`property num_chunks`

The total number of chunks that make up the hypernet output.

This attribute simply corresponds to `np.sum(num_per_chunk)`.

Type
int

`training: bool`

CHAPTER

THREE

HYPERPARAMETER SEARCHES

Contents

3.1 A general framework to perform hyperparameter searches on single- and multi-GPU systems

Note, we currently only support simple grid searches.

3.1.1 How to run a hyperparameter search

The main script in this package is *hypnettorch.hpsearch.hpsearch*.

```
$ python -m hypnettorch.hpsearch.hpsearch --help
```

Though, before being able to run a hyperparameter search, the search grid has to be configured. Therefore, your simulation has its own implementation of the configuration file *hypnettorch.hpsearch.hpsearch_config_template*. Please refer to the corresponding documentation to obtain information on how to configure a hyperparameter search.

Execute on a single- or multi-GPU system without job scheduling

The simplest way of execution is to run all hyperparameter configurations sequentially in the foreground. For instance, on a computer without GPUs, you could start the hpsearch on the CPU as follows

```
$ python -m hypnettorch.hpsearch.hpsearch --visible_gpus=-1
```

Though, assuming that your simulations automatically run on a visible GPU, you can also apply this sequential foreground execution to a GPU of your choice (e.g., GPU 2):

```
$ CUDA_VISIBLE_DEVICES=2 python -m hypnettorch.hpsearch.hpsearch --visible_gpus=-1
```

Alternatively, the hpsearch may assign GPU ressources to jobs. In this case, multiple hyperparameter configurations may run in parallel (on multiple GPUs as well as multiple runs per GPU). For this operation mode, you are required to install the package GPUtil.

Please carefully study the arguments of the hpsearch.

```
$ python -m hypnettorch.hpsearch.hpsearch --help
```

Assume you may want to run your search on GPUs 0,1,2,7 and that there should be a hard limit of 5 jobs assigned to a GPU by the hpsearch (which you decide based on available CPU and RAM ressources). Note, option `--max_num_jobs_per_gpu` currently does not account for other processes that may be running on the GPU which are not assigned by this hpsearch. In addition, a run may only be assigned to a GPU if at maximum 75% of its memory is in use and its compute utilization is maximally at 60%. Since runs take some time to properly startup and allocate GPU ressources, you additionally specify argument `--sim_startup_time`. Every time a job is assigned to a GPU, this time has to pass before a new job may be assigned (such that the first job had time to acquire GPU memory and compute ressources)

```
$ python -m hypnettorch.hpsearch.hpsearch --visible_gpus=0,1,2,7 --max_num_jobs_per_
↪gpu=5 --allowed_memory=0.75 --allowed_load=0.6 --sim_startup_time=30
```

Execute on a cluster with IBM Platform LSF

You may also run the hpsearch on a cluster that uses the IBM Platform LSF job scheduler. In this case, you have to install the package bsub. To tell the hpsearch that should schedule jobs via `bsub`, simply append the options `--run_cluster --scheduler=lsf`. Here is an example call:

```
$ bsub -n 1 -W 120:00 -e hpsearch_mysim.err -o hpsearch_mysim.out -R "rusage[mem=8000]"␣
↪python -m hypnettorch.hpsearch.hpsearch --grid_module=my_hpsearch_config --run_cluster␣
↪--scheduler=lsf --num_jobs=50 --num_hours=24 --num_searches=1000 --resources="\
↪"rusage[mem=8000, ngpus_excl_p=1]\""
```

In the example above, the hpsearch should run for 120 hours on the cluster, requiring 8GB of RAM during that time. Individual jobs will run for 24 hours. The hpsearch will maximally explore 1000 hyperparameter configurations. At most 50 jobs will be scheduled in parallel (new jobs will be scheduled as soon as old ones finished until the hard limit of 1000 runs is reached). Each job will require 1 GPU and 8GB of RAM.

Execute on a cluster with Slurm Workload Manager

The hpsearch can also be run on a cluster with the SLURM job scheduler via the arguments `--run_cluster --scheduler=slurm`. Therefore, simply create a job script `my_hpsearch.sh` for the hpsearch as follows

```
#!/bin/bash
#SBATCH --job-name=hpsearch
#SBATCH --output=hpsearch_%j.out
#SBATCH --error=hpsearch_%j.err
#SBATCH --time=24:00:00
#SBATCH --mem=8G
python -m hypnettorch.hpsearch.hpsearch --grid_module=my_hpsearch_config --run_cluster --
↪scheduler=slurm --slurm_mem=8G --slurm_gres=gpu:1 --num_jobs=25 --num_hours=4
```

The hpsearch can be executed via the command:

```
$ sbatch my_hpsearch.sh
```

Execute on a cluster with unsupported job scheduler

Unfortunately, you can only execute the hpsearch on a cluster with unsupported job scheduler in the sequential foreground mode via `--visible_gpus=-1`. For instance, on a cluster running the SLURM job scheduler (note, SLURM is supported, see above) you can run the hpsearch in sequential forground mode via a script `my_hpsearch.sh`:

```
#!/bin/bash
#SBATCH --job-name=hpsearch
#SBATCH --output=hpsearch_%j.out
#SBATCH --error=hpsearch_%j.err
#SBATCH --time=120:00:00
#SBATCH --mem=8G
#SBATCH --gres gpu:1
python -m hypnettorch.hpsearch.hpsearch --grid_module=my_hpsearch_config --visible_gpus=-
↪1
```

Note, in this case, you request the ressources required for your jobs for the hpsearch itself. Now, you could execute the hpsearch via

```
$ sbatch my_hpsearch.sh
```

3.1.2 Postprocessing

The post processing script *hypnettorch.hpsearch.hpsearch_postprocessing* is currently very rudimentary. Its most important task is to make sure that the results of all completed runs are listed in a CSV file (note, that the hpsearch might be killed prematurely while some jobs are still running).

Please checkout

```
$ python3 -m hypnettorch.hpsearch.hpsearch_postprocessing --help
```

3.1.3 How to use this framework with your simulation

In order to utilize the scripts in this subpackage, you have to create a copy of the template *hypnettorch.hpsearch.hpsearch_config_template* and fill the template with content as described inside the module. For instance, see probabilistic.prob_mnist.hpsearch_config_split_bbb as an example.

Additionally, you need to make sure that your simulation has a command-line option like `--out_dir` (that specifies the output directory) and that your simulation writes a performance summary file, that can be used to evaluate simulations.

Gather random seeds for a given experiment

This script can be used to gather random seeds for a given configuration. Thus, it is intended to test the robustness of this certain configuration.

The configuration can either be provided directly, or the path to a simulation output folder or hyperparameter search output folder is provided. A simulation output folder is recognized by the file `config.pickle` which contains the *configuration*, i.e., all command-line arguments (cf. function `hypnettorch.sim_utils.setup_environment()`). If a hyperparameter search output folder (cf. *hypnettorch.hpsearch.hpsearch*) is provided, the best run will be selected.

Example 1: Assume you are in the simulation directory and want to start the random seed gathering from there for a simulation in folder `./out/example_run`. Note, we assume here that the base run in `./out/example_run` finished successfully and can already be used as 1 random seed.

```
$ python -m hypnettorch.hpsearch.gather_random_seeds --grid_module=my_hpsearch_config --
↪run_dir=./out/example_run --num_seeds=10 | tee /dev/tty | awk 'END{print}' | xargs␣
↪bash -c 'echo --grid_module=$0 --grid_config=$1 --force_out_dir --dont_force_new_dir --
↪out_dir=$2' | xargs python -m hypnettorch.hpsearch
```

Example 2: Alternatively, the hpsearch can be started directly via the option `--start_gathering`.

```
$ python -m hypnettorch.hpsearch.gather_random_seeds --grid_module=my_hpsearch_config --
↪run_dir=./out/example_run --num_seeds=4 --start_gathering --config_name=example_run_
↪seed_gathering
```

Example 3: An example instantiation of this script can be found in module probabilistic.regression.gather_seeds_bbb.

hypnettorch.hpsearch.gather_random_seeds.**build_grid_and_conditions**(*cmd_args*, *config*, *seeds_list*)

Build the hpconfig for the random seed gathering.

Parameters

- **cmd_args** – CLI arguments of this script.
- **config** – The config to be translated into a search grid.
- **seeds_list** (*list*) – The random seeds to be gathered.

(tuple): Tuple containing:

- **grid** (dict): The search grid.
- **conditions** (list): Constraints for the search grid.

hypnettorch.hpsearch.gather_random_seeds.**get_best_hpsearch_config**(*out_dir*)

Load the config file from the best run of a hyperparameter search.

This file loads the results of the hyperparameter search, and select the configuration that lead to the best performance score.

Parameters
out_dir (*str*) – The path to the hpsearch result folder.

Returns

Tuple containing:

- **config**: The config of the best run.
- **best_out_dir**: The path to the best run.

Return type
(tuple)

hypnettorch.hpsearch.gather_random_seeds.**get_hpsearch_call**(*cmd_args*, *num_seeds*, *grid_config*, *hpsearch_dir=None*)

Generate the command line for the hpsearch.

Parameters

- **cmd_args** – The command line arguments.
- **num_seeds** (*int*) – Number of searches.
- **grid_config** (*str*) – Location of search grid.
- **hpsearch_dir** (*str, optional*) – Where the hpsearch should write its results to.

Returns
The command line to be executed.

Return type
(str)

hypnettorch.hpsearch.gather_random_seeds.**get_single_run_config**(*out_dir*)

Load the config file from a specified experiment.

Parameters
out_dir (*str*) – The path to the experiment.

Returns
The Namespace object containing argument names and values.

hypnettorch.hpsearch.gather_random_seeds.**run**(*grid_module=None*, *results_dir='./out/random_seeds'*, *config=None*, *ignore_kwds=None*, *forced_params=None*, *summary_keys=None*, *summary_sem=False*, *summary_precs=None*, *hpmod_path=None*)

Run the script.

Parameters

- **grid_module** (*str, optional*) – Name of the reference module which contains the hyperparameter search config that can be modified to gather random seeds.
- **results_dir** (*str, optional*) – The path where the hpsearch should store its results.
- **config** – The Namespace object containing argument names and values. If provided, all random seeds will be gathered from zero, with no reference run.
- **ignore_kwds** (*list, optional*) – A list of keywords in the config file to exclude from the grid.
- **forced_params** (*dict, optional*) – Dict of key-value pairs specifying hyperparameter values that should be fixed across runs.
- **summary_keys** (*list, optional*) – If provided, those mean and std of those summary keys will be written by function *write_seeds_summary()*. Otherwise, the performance key defined in `grid_module` will be used.
- **summary_sem** (*bool*) – Whether SEM or SD should be calculated in function *write_seeds_summary()*.
- **summary_precs** (*list or int, optional*) – The precision with which the summary statistics according to `summary_keys` should be listed.
- **hpmod_path** (*str, optional*) – If the hpsearch doesn't reside in the same directory as the calling script, then we need to know from where to start the hpsearch.

hypnettorch.hpsearch.gather_random_seeds.**write_seeds_summary**(*results_dir*, *summary_keys*, *summary_sem*, *summary_precs*, *ret_seeds=False*, *summary_fn=None*, *seeds_summary_fn='seeds_summary_text.txt'*)

Write the MEAN and STD (resp. SEM) while aggregating all seeds to text file.

Parameters

- **results_dir** (*str*) – The results directory.
- **summary_keys** (*list*) – See argument `summary_keys` of function *run()*.
- **summary_sem** (*bool*) – See argument `summary_sem` of function *run()*.
- **summary_precs** (*list or int, optional*) – See argument `summary_precs` of function *run()*.
- **summary_fn** (*str, optional*) – If given, this will determine the name of the summary file within individual runs.
- **seeds_summmary_fn** (*str, optional*) – The name to give to the summary file across all seeds.
- **ret_seeds** (*bool, optional*) – If activated, the random seeds of all considered runs are returned as a list.

Hyperparameter Search Configuration File

hypnettorch.hpsearch.hpsearch_config_template.conditions	Define exceptions for the grid search.
hypnettorch.hpsearch.hpsearch_config_template.grid	Parameter grid for grid search.

Note, this is just a template for a hyperparameter configuration and not an actual source file.

A configuration file for our custom hyperparameter search script *hypnettorch.hpsearch.hpsearch*. To setup a configuration file for your simulation, simply create a copy of this template and follow the instructions in this file to fill all defined attributes.

Once the configuration is setup for your simulation, you simply need to modify the fields *grid* and *conditions* to prepare a new grid search.

Note, if you are implementing this template for the first time, you also have to modify the code below the "DO NOT CHANGE THE CODE BELOW" section. Normal users may not change the code below this heading.

hypnettorch.hpsearch.hpsearch_config_template.**conditions = [({'option1': [1]}, {'option2': [-1]})]**

Define exceptions for the grid search.

Sometimes, not the whole grid should be searched. For instance, if an *SGD* optimizer has been chosen, then it doesn't make sense to search over multiple *beta2* values of an Adam optimizer. Therefore, one can specify special conditions or exceptions. Note* all conditions that are specified here will be enforced. Thus, **they overwrite the** *grid* **options above**.

How to specify a condition? A condition is a key value tuple: whereas as the key as well as the value is a dictionary in the same format as in the *grid* above. If any configurations matches the values specified in the "key" dict, the values specified in the "values" dict will be searched instead.

Note, if arguments are commented out above but appear in the conditions, the condition will be ignored.

Also keep in mind, that the hpsearch is not checking for conflicting conditions and they are enforced sequentially. For instance, assume condition 2 would change commands such that condition 1 would fire again. But condition 1 is never tested again, so these commands would make it into the final hpsearch (unless later conditions modify them again).

hypnettorch.hpsearch.hpsearch_config_template.**grid = {'flag_option': [False, True], 'float_option': [0.5, 1.0], 'string_option': ['"example string"', '"another string"']}**

Parameter grid for grid search.

Define a dictionary with parameter names as keys and a list of values for each parameter. For flag arguments, simply use the values `[False, True]`. Note, the output directory is set by the hyperparameter search script. Therefore, it always assumes that the argument *–out_dir* exists and you **should not** add *out_dir* to this *grid*!

Example

```
grid = {'option1': [10], 'option2': [0.1, 0.5],
        'option3': [False, True]}
```

This dictionary would correspond to the following 4 configurations:

```
python3 SCRIPT_NAME.py --option1=10 --option2=0.1
python3 SCRIPT_NAME.py --option1=10 --option2=0.5
python3 SCRIPT_NAME.py --option1=10 --option2=0.1 --option3
python3 SCRIPT_NAME.py --option1=10 --option2=0.5 --option3
```

If fields are commented out (missing), the default value is used. Note, that you can specify special *conditions* below.

Hyperparameter Search - Postprocessing

A postprocessing for a hyperparameter search that has been executed via the script *hypnettorch.hpsearch.hpsearch*.

Hyperparameter Search Script

A very simple hyperparameter search. The results will be gathered as a CSV file.

Here is an example on how to start an hyperparameter search on a cluster using `bsub`:

```
$ bsub -n 1 -W 48:00 -e hpsearch.err -o hpsearch.out \
  -R "rusage[mem=8000]" \
  python -m hypnettorch.hpsearch.hpsearch --run_cluster --num_jobs=20
```

For more demanding jobs (e.g., ImageNet), one may request more resources:

```
$ bsub -n 1 -W 96:00 -e hpsearch.err -o hpsearch.out \
  -R "rusage[mem=16000]" \
  python -m hypnettorch.hpsearch.hpsearch --run_cluster --num_jobs=20 \
  --num_hours=48 --resources="\"rusage[mem=8000, ngpus_excl_p=1]\""
```

Please fill in the grid parameters in the corresponding config file (see command line argument *grid_module*).

hypnettorch.hpsearch.hpsearch.**hpsearch_cli_arguments**(*parser*, *show_num_searches=True*, *show_out_dir=True*, *dout_dir='./out/hyperparam_search'*, *show_grid_module=True*)

The CLI arguments of the hpsearch.

hypnettorch.hpsearch.hpsearch.**run**(*argv=None*, *dout_dir='./out/hyperparam_search'*)

Run the hyperparameter search script.

Parameters

- **argv** (*list, optional*) – If provided, it will be treated as a list of command-line argument that is passed to the parser in place of `sys.argv`.
- **dout_dir** (*str, optional*) – The default value of command-line option `--out_dir`.

Returns
The path to the CSV file containing the results of this search.

Return type
(str)

CHAPTER

FOUR

MAIN NETWORKS

Contents

Note: All main networks should inherit from the abstract class *hypnettorch.mnets.mnet_interface.MainNetInterface* to provide a consistent interface for users.

4.1 Bidirectional Recurrent Neural Network

This module implements a bidirectional recurrent neural networt (BiRNN). To realize recurrent layers, it utilizes class `mnets.simple_rnn.SimpleRNN`. Hence different kinds of BiRNNs can be realized, such as Elman-type BiRNNs and BiLSTMs. In particular, this class implements the BiRNN in the following manner. Given an input $x_{1:T}$, the forward RNN is run to produce hidden states $\hat{h}^{(f)}_{1:T}$ and the backward RNN is run to produce states $\hat{h}^{(b)}_{1:T}$.

These hidden states are concatenated to produce the final hidden state which is the output of the recurrent layer(s): $h_t = \text{concat}(\hat{h}^{(f)}_t, \hat{h}^{(b)}_t)$.

Those inputs are subsequently processed by an instance of class `mnets.mlp.MLP` to produce the final network outputs.

class hypnettorch.mnets.bi_rnn.**BiRNN**(*rnn_args={}*, *mlp_args=None*, *preprocess_fct=None*, *no_weights=False*, *verbose=True*)

Bases: Module, *MainNetInterface*

Implementation of a bidirectional RNN.

Note: The output is non-linear if the last layer is recurrent! Otherwise, logits are returned (cmp. attribute `mnets.mnet_interface.MainNetInterface.has_fc_out`).

Example

Here is an example instantiation of a BiLSTM with a single bidirectional layer of dimensionality 256, assuming 100 dimensional inputs and 10 dimensional outputs.

```
net = BiRNN(rnn_args={'n_in': 100, 'rnn_layers': [256],
                      'use_lstm': True, 'fc_layers_pre': [],
                      'fc_layers': []},
            mlp_args={'n_in': 512, 'n_out': 10,
                      'hidden_layers': []},
            no_weights=False)
```

Parameters

- **rnn_args** (*dict or list*) – A dictionary of arguments for an instance of class `mnets.simple_rnn.SimpleRNN`. These arguments will be used to create two instances of this class, one representing the forward RNN and one the backward RNN.

  Note, each of these instances may contain multiple layers, even non-recurrent layers. The outputs of such an instance are considered the hidden activations $\hat{h}_{1:T}^{(f)}$ or $\hat{h}_{1:T}^{(b)}$, respectively.

  To realize multiple bidirectional layers (which in itself can be multi-layer RNNs), one may provide a list of dictionaries. Each entry in such list will be used to generate a single bidirectional layer (i.e., consisting of two instances of class `mnets.simple_rnn.SimpleRNN`). Note, the input size of each new layer has to be twice the size of $\hat{h}_t^{(f)}$ from the previous layer.

- **mlp_args** (*dict, optional*) – A dictionary of arguments for class `mnets.mlp.MLP`. The input size of such an MLP should be twice the size of $\hat{h}_t^{(f)}$. If `None`, then the output of the last bidirectional layer is considered the output of the network.

- **preprocess_fct** (*func, optional*) – A function handle can be provided, that will process inputs `x` passed to the method *forward()*. An example usecase could be the translation or selection of word embeddings.

  The function handle must have the signature: `preprocess_fct(x, seq_lengths=None)`. See the corresponding argument descriptions of method *forward()*.The function is expected to return the preprocessed `x`.

- **no_weights** (*bool*) – See parameter `no_weights` of class `mnets.mlp.MLP`.

- **verbose** (*bool*) – See parameter `verbose` of class `mnets.mlp.MLP`.

Initializes internal Module state, shared by both nn.Module and ScriptModule.

distillation_targets()

Targets to be distilled after training.

See docstring of abstract super method mnets.mnet_interface.MainNetInterface.distillation_targets().

forward(*x*, *weights=None*, *distilled_params=None*, *condition=None*, *seq_lengths=None*)

Compute the output y of this network given the input x.

Note: If constructor argument preprocess_fct was set, then all inputs x are first processed by this function.

Parameters

- **(....)** – See docstring of method mnets.mnet_interface.MainNetInterface.forward(). We provide some more specific information below.
- **weights** (*list or dict*) – See argument weights of method mnets.mlp.MLP.forward().
- **distilled_params** – Will only be passed to the underlying instance of class mnets.mlp.MLP
- **condition** (*int or dict, optional*) – If provided, then this argument will be passed as argument ckpt_id to the method utils.context_mod_layer.ContextModLayer.forward().

  When providing as dict, see argument condition of method mnets.mlp.MLP.forward() for more details.
- **seq_lengths** (*numpy.ndarray, optional*) – List of sequence lengths. The length of the list has to match the batch size of inputs x. The entries will correspond to the unpadded sequence lengths. If this option is provided, then the bidirectional layers will reverse its input sequences according to the unpadded sequence lengths.

  ### Example

  x = [[a,b,0,0], [a,b,c,0]].T. If seq_lengths = [2, 3] if provided, then the reverse sequences [[b,a,0,0], [c,b,a,0]].T are fed into the first bidirectional layer (and similarly for all subsequent bidirectional layers). Otherwise reverse sequences [[0, 0,b,a], [0,c,b,a]].T are used.

  Caution: If this option is not provided but padded input sequences are used, the output of a bidirectional layer will depent on the padding. I.e., different padding lengths will lead to different results.

Returns

Where the tuple is containing:

- **output** (torch.Tensor): The output of the network.
- **hidden** (list): None - not implemented yet.

Return type
(torch.Tensor or tuple)

`get_cm_weights()`

Get internal maintained weights that are associated with context- modulation.

Returns
List of weights from `mnets.mnet_interface.MainNetInterface.internal_params` that are belonging to context-mod layers.

Return type
(list)

`get_non_cm_weights()`

Get internal weights that are not associated with context-modulation.

Returns
List of weights from `mnets.mnet_interface.MainNetInterface.internal_params` that are not belonging to context-mod layers.

Return type
(list)

`init_hh_weights_orthogonal()`

Initialize hidden-to-hidden weights orthogonally.

This method will call method `mnets.simple_rnn.SimpleRNN.init_hh_weights_orthogonal()` of all internally maintained instances of class `mnets.simple_rnn.SimpleRNN`.

`property num_rec_layers`

See attribute `mnets.simple_rnn.SimpleRNN.num_rec_layers`. Total number of recurrent layer, where each bidirectional layer consists of at least two recurrent layers (forward and backward layer).

Type
int

`property preprocess_fct`

See constructor argument `preprocess_fct`.

Setter
The setter may only be called before the first call of the *forward()* method.

Type
func

`training: bool`

`property use_lstm`

See attribute `mnets.simple_rnn.SimpleRNN.use_lstm`.

Type
bool

4.2 A bio-plausible convolutional network for CIFAR

The module `mnets.bio_conv_net` implements a simple biologically-plausible network with convolutional and fully-connected layers. The bio-plausibility arises through the usage of conv-layers without weight sharing, i.e., layers from class `utils.local_conv2d_layer.LocalConv2dLayer`. The network specification has been taken from the following paper

> Bartunov et al., "Assessing the Scalability of Biologically-Motivated Deep Learning Algorithms and Architectures", NeurIPS 2018.

in which this kind of network has been termed "locally-connected network".

In particular, we consider the network architecture specified in table 3 on page 13 for the CIFAR dataset.

hypnettorch.mnets.bio_conv_net.BioConvNet([...])	Implementation of a locally-connected network for CIFAR.

class hypnettorch.mnets.bio_conv_net.**BioConvNet**(*in_shape=(32, 32, 3)*, *num_classes=10*, *no_weights=False*, *init_weights=None*, *use_context_mod=False*, *context_mod_inputs=False*, *no_last_layer_context_mod=False*, *context_mod_no_weights=False*, *context_mod_post_activation=False*, *context_mod_gain_offset=False*, *context_mod_gain_softplus=False*, *context_mod_apply_pixel_wise=False*)

Bases: *Classifier*

Implementation of a locally-connected network for CIFAR.

The network consists of 3 bio-plausible convolutional layers (using class `utils.local_conv2d_layer.LocalConv2dLayer`) followed by two fully-connected layers.

Assume conv layers are specified by the tuple `(K x K, C, S, P)`, where `K` denotes the kernel size, `C` the number of channels, `S` the stride and `P` the padding. The network is defined as follows

- Bio-conv layer (5 x 5, 64, 2, 0)
- Bio-conv layer (5 x 5, 128, 2, 0)
- Bio-conv layer (3 x 3, 256, 1, 1)
- FC layer with 1024 outputs
- FC layer with 10 outputs

Note, the padding for the first two convolutional layers was not specified in the paper, so we just assumed it to be zero.

The **network output will be linear**, so we do not apply the softmax inside the *forward()* method.

Note, the paper states that `tanh` was used in all networks as non-linearity. Therefore, we use this non-linearity too.

Parameters

- **in_shape** – The shape of an input sample.

Note: We assume the Tensorflow format, where the last entry denotes the number of channels.

- **num_classes** – The number of output neurons.
- **no_weights** (*bool*) – If set to `True`, no trainable parameters will be constructed, i.e., weights are assumed to be produced ad-hoc by a hypernetwork and passed to the *forward()* method.
- **init_weights** (*optional*) – This option is for convinience reasons. The option expects a list of parameter values that are used to initialize the network weights. As such, it provides a convinient way of initializing a network with a weight draw produced by the hypernetwork.

  Note, internal weights (see `mnets.mnet_interface.MainNetInterface.weights`) will be affected by this argument only.
- **use_context_mod** (*bool*) – Add context-dependent modulation layers `utils.context_mod_layer.ContextModLayer` after the linear computation of each layer.
- **context_mod_inputs** (*bool*) – Whether context-dependent modulation should also be applied to network intpus directly. I.e., assume $\mathbf{x}$ is the input to the network. Then the first network operation would be to modify the input via $\mathbf{x} \cdot \mathbf{g} + \mathbf{s}$ using context- dependent gain and shift parameters.

  Note: Argument applies only if `use_context_mod` is `True`.
- **no_last_layer_context_mod** (*bool*) – If `True`, context-dependent modulation will not be applied to the output layer.

  Note: Argument applies only if `use_context_mod` is `True`.
- **context_mod_no_weights** (*bool*) – The weights of the context-mod layers (`utils.context_mod_layer.ContextModLayer`) are treated independently of the option `no_weights`. This argument can be used to decide whether the context-mod parameters (gains and shifts) are maintained internally or externally.

  Note: Check out argument `weights` of the *forward()* method on how to correctly pass weights to the network that are externally maintained.
- **context_mod_post_activation** (*bool*) – Apply context-mod layers after the activation function (`activation_fn`) in hidden layer rather than before, which is the default behavior.

  Note: This option only applies if `use_context_mod` is `True`.

  Note: This option does not affect argument `context_mod_inputs`.

  Note: Note, there is no non-linearity applied to the output layer, such that this argument has

no effect there.

- **context_mod_gain_offset** (*bool*) – Activates option `apply_gain_offset` of class `utils.context_mod_layer.ContextModLayer` for all context-mod layers that will be instantiated.
- **context_mod_gain_softplus** (*bool*) – Activates option `apply_gain_softplus` of class `utils.context_mod_layer.ContextModLayer` for all context-mod layers that will be instantiated.
- **context_mod_apply_pixel_wise** (*bool*) – If `False`, the context-dependent modulation applies a scalar gain and shift to all feature maps in the output of a convolutional layer. When activating this option, the gain and shift will be a per-pixel parameter in all feature maps.

  To be more precise, consider the output of a convolutional layer of shape `[C,H,W]`. If `False`, there will be `C` gain and shift parameters for such a layer. Upon activating this option, the number of gain and shift parameters for such a layer will increase to `C x H x W`.

Initialize the network.

Parameters

- **num_classes** – The number of output neurons.
- **verbose** – Allow printing of general information about the generated network (such as number of weights).

distillation_targets()

Targets to be distilled after training.

See docstring of abstract super method `mnets.mnet_interface.MainNetInterface.distillation_targets()`.

This network does not have any distillation targets.

Returns
`None`

forward(*x*, *weights=None*, *distilled_params=None*, *condition=None*, *collect_activations=False*)

Compute the output y of this network given the input x.

Parameters

- **(....)** – See docstring of method `mnets.mnet_interface.MainNetInterface.forward()`. We provide some more specific information below.
- **x** – Input image.

  Note: We assume the Tensorflow format, where the last entry denotes the number of channels.

- **weights** (*list or dict*) – If a list of parameter tensors is given and context modulation is used (see argument `use_context_mod` in constructor), then these parameters are interpreted as context- modulation parameters if the length of `weights` equals `2*len(net.context_mod_layers)`. Otherwise, the length is expected to be equal to the length of the attribute `mnets.mnet_interface.MainNetInterface.param_shapes`.

  Alternatively, a dictionary can be passed with the possible keywords `internal_weights` and `mod_weights`. Each keyword is expected to map onto a list of tensors. The keyword `internal_weights` refers to all weights of this network except for the weights of the

context-modulation layers. The keyword `mod_weights`, on the other hand, refers specifically to the weights of the context-modulation layers. It is not necessary to specify both keywords.

- **condition** (*int, optional*) – Will be passed as argument `ckpt_id` to the method `utils.context_mod_layer.ContextModLayer.forward()` for all context-mod layers in this network.
- **collect_activations** (*bool, optional*) – If one wants to return the activations in the network. This information can be used for credit assignment later on, in case an alternative to PyTorch its `torch.autograd` should be used.

Returns

Tuple containing:

- **y**: The output of the network.
- **layer_activation** (optional): The activations of the network. Only returned if `collect_activations` was set to `True`. The list will contain the activations of all convolutional and linear layers.

Return type
(`torch.Tensor` or tuple)

`training: bool`

4.3 Interface for Classifiers

A general interface for main networks used in classification tasks. This abstract base class also provides a collection of static helper functions that are useful in classification problems.

`class hypnettorch.mnets.classifier_interface.Classifier(num_classes, verbose)`

Bases: `Module`, *MainNetInterface*

A general interface for classification networks.

Initialize the network.

Parameters

- **num_classes** – The number of output neurons.
- **verbose** – Allow printing of general information about the generated network (such as number of weights).

`static accuracy(y, t)`

Computing the accuracy between predictions y and targets t. We assume that the argmax of t results in labels as described in the docstring of method "cross_entropy_loss".

Parameters

- **y** – Outputs from the main network.
- **t** – Targets in form of soft labels or 1-hot encodings.

Returns
Relative prediction accuracy on the given batch.

static knowledge_distillation_loss(*logits*, *target_logits*, *target_mapping=None*, *device=None*, *T=2.0*)

Compute the knowledge distillation loss as proposed by

Hinton et al., "Distilling the Knowledge in a Neural Network", NIPS Deep Learning and Representation Learning Workshop, 2015. http://arxiv.org/abs/1503.02531

Parameters

- **logits** – Unscaled outputs from the main network, i.e., activations of the last hidden layer (unscaled logits).
- **target_logits** – Target logits, i.e., activations of the last hidden layer (unscaled logits) from the target model. Note, we won't detach "target_logits" from the graph. Make sure, that you do this before calling this method.
- **target_mapping** – In continual learning, it might be that the output layer size of a model is growing. Thus, it could be that the model providing the `target_logits` has a smaller output size than the current model providing the `logits`. Therefore, one has to provide a mapping, which is a list of indices for `logits` that state which activations in `logits` have a corresponding target in `target_logits`. For instance, if the output layer size just increased by 1 through appending a new output neuron to the current model, the mapping would simply be: `target_mapping = list(range(target_logits.shape[1]))`.
- **device** – Current PyTorch device. Only needs to be specified if "target_mapping" is given.
- **T** – Softmax temperature.

Returns
Knowledge Distillation (KD) loss.

static logit_cross_entropy_loss(*h*, *t*, *reduction='mean'*)

Compute cross-entropy loss for given predictions and targets. Note, we assume that the argmax of the target vectors results in the correct label.

Parameters

- **h** – Unscaled outputs from the main network, i.e., activations of the last hidden layer (unscaled logits).
- **t** – Targets in form os soft labels or 1-hot encodings.
- **reduction** (*str*) – The reduction method to be passed to `torch.nn.functional.cross_entropy()`.

Returns
Cross-entropy loss computed on logits h and labels extracted from target vector t.

property num_classes

Number of output neurons.

Type
int

static num_hyper_weights(*dims*)

The number of weights that have to be predicted by a hypernetwork.

Deprecated since version 1.0: Please use method `mnets.mnet_interface.MainNetInterface.shapes_to_num_weights()` instead.

Parameters
dims – For instance, the attribute `hyper_shapes`.

Returns
(int)

`static softmax_and_cross_entropy(h, t, reduction_sum=False)`

Compute the cross entropy from logits, allowing smoothed labels (i.e., this function does not require 1-hot targets).

Parameters

- **h** – Unscaled outputs from the main network, i.e., activations of the last hidden layer (unscaled logits).
- **t** – Targets in form os soft labels or 1-hot encodings.

Returns
Cross-entropy loss computed on logits h and given targets t.

`training: bool`

4.4 LeNet

This module contains a general classifier template and a LeNet-like network to classify either MNIST or CIFAR-10 images. The network is implemented in a way that it might not have trainable parameters. Instead, the network weights would have to be passed to the `forward` method. This makes the usage of a hypernetwork (a network that generates the weights of another network) particularly easy.

`class hypnettorch.mnets.lenet.LeNet(in_shape=(28, 28, 1), num_classes=10, verbose=True, arch='mnist_large', no_weights=False, init_weights=None, dropout_rate=-1, **kwargs)`

Bases: *Classifier*

The network consists of two convolutional layers followed by two fully- connected layers. See implementation for details.

LeNet was originally introduced in

> "Gradient-based learning applied to document recognition", LeCun et al., 1998.

Though, the implementation provided here has several difference compared to the original LeNet architecture (e.g., the LeNet-5 architecture):

- There is no special connectivity map before the second convolutional layer as described by table 1 in the original paper.
- The dimensions of layers and their activation functions are dfferent.
- The original LeNet-5 has a third fully connected layer with 1x1 kernels.

We mainly use this modified LeNet architecture for MNIST:

- A small architecture with only 21,840 weights.
- A larger architecture with 431,080 weights.

Both of these architectures are typically used for MNIST nowadays.

Note, a variant of this architecture is also used for CIFAR-10, e.g. in

"Bayesian Convolutional Neural Networks with Bernoulli Approximate Variational Inference", Gal et al., 2015.

and

"Multiplicative Normalizing Flows for Variational Bayesian Neural Networks", Louizos et al., 2017.

In both these works, the dimensions of the weight parameters are:

```
main_dims=[[192,3,5,5],[192],[192,192,5,5],[192],[1000,4800],
           [1000],[10,1000],[10]],
```

which is an architecture with 5,747,394 weights. Note, the authors used dropout in different configurations, e.g., after each layer, only after the fully-connected layer or no dropout at all.

Parameters

- **in_shape** (*tuple or list*) – The shape of an input sample.

  Note: We assume the Tensorflow format, where the last entry denotes the number of channels.

- **num_classes** (*int*) – The number of output neurons.
- **verbose** (*bool*) – Allow printing of general information about the generated network (such as number of weights).
- **arch** (*str*) – The architecture to be employed. The following options are available:
  - `'mnist_small'`: A small LeNet with 21,840 weights suitable for MNIST
  - `'mnist_large'`: A larger LeNet with 431,080 weights suitable for MNIST
  - `'cifar'`: A huge LeNet with 5,747,394 weights designed for CIFAR-10.
- **no_weights** (*bool*) – If set to `True`, no trainable parameters will be constructed, i.e., weights are assumed to be produced ad-hoc by a hypernetwork and passed to the *forward()* method.
- **init_weights** (*optional*) – This option is for convinience reasons. The option expects a list of parameter values that are used to initialize the network weights. As such, it provides a convinient way of initializing a network with a weight draw produced by the hypernetwork.
- **dropout_rate** (*float*) – If `-1`, no dropout will be applied. Otherwise a number between 0 and 1 is expected, denoting the dropout rate.

  Dropout will be applied after the convolutional layers (before pooling) and after the first fully-connected layer (after the activation function).
- ****kwargs** – Keyword arguments regarding context modulation. This class can process the same context-modulation related arguments as class `mnets.mlp.MLP`. One may additionally specify the argument `context_mod_apply_pixel_wise` (see class `mnets.resnet.ResNet`).

Initialize the network.

Parameters

- **num_classes** – The number of output neurons.
- **verbose** – Allow printing of general information about the generated network (such as number of weights).

distillation_targets()

Targets to be distilled after training.

See docstring of abstract super method mnets.mnet_interface.MainNetInterface.distillation_targets().

This network does not have any distillation targets.

Returns
None

forward(*x*, *weights=None*, *distilled_params=None*, *condition=None*)

Compute the output y of this network given the input x.

Parameters

- **(....)** – See docstring of method mnets.mnet_interface.MainNetInterface.forward(). We provide some more specific information below.
- **weights** (*list or dict*) – See argument weights of method mnets.mlp.MLP.forward().
- **condition** (*int, optional*) – If provided, then this argument will be passed as argument ckpt_id to the method utils.context_mod_layer.ContextModLayer.forward().

Returns
The output of the network.

Return type
(torch.Tensor)

training: bool

4.5 Multi-Layer Perceptron

Implementation of a fully-connected neural network.

An example usage is as a main model, that doesn't include any trainable weights. Instead, weights are received as additional inputs. For instance, using an auxilliary network, a so called hypernetwork, see

Ha et al., "HyperNetworks", arXiv, 2016, https://arxiv.org/abs/1609.09106

class hypnettorch.mnets.mlp.**MLP**(*n_in=1*, *n_out=1*, *hidden_layers=(10, 10)*, *activation_fn=ReLU()*, *use_bias=True*, *no_weights=False*, *init_weights=None*, *dropout_rate=-1*, *use_spectral_norm=False*, *use_batch_norm=False*, *bn_track_stats=True*, *distill_bn_stats=False*, *use_context_mod=False*, *context_mod_inputs=False*, *no_last_layer_context_mod=False*, *context_mod_no_weights=False*, *context_mod_post_activation=False*, *context_mod_gain_offset=False*, *context_mod_gain_softplus=False*, *out_fn=None*, *verbose=True*)

Bases: Module, *MainNetInterface*

Implementation of a Multi-Layer Perceptron (MLP).

This is a simple fully-connected network, that receives input vector **x** and outputs a vector **y** of real values.

The output mapping does not include a non-linearity by default, as we wanna map to the whole real line (but see argument out_fn).

Parameters

- **n_in** (*int*) – Number of inputs.
- **n_out** (*int*) – Number of outputs.
- **hidden_layers** (*list or tuple*) – A list of integers, each number denoting the size of a hidden layer.
- **activation_fn** – The nonlinearity used in hidden layers. If `None`, no nonlinearity will be applied.
- **use_bias** (*bool*) – Whether layers may have bias terms.
- **no_weights** (*bool*) – If set to `True`, no trainable parameters will be constructed, i.e., weights are assumed to be produced ad-hoc by a hypernetwork and passed to the *forward()* method.
- **init_weights** (*optional*) – This option is for convinience reasons. The option expects a list of parameter values that are used to initialize the network weights. As such, it provides a convinient way of initializing a network with a weight draw produced by the hypernetwork.

  Note, internal weights (see `mnets.mnet_interface.MainNetInterface.weights`) will be affected by this argument only.
- **dropout_rate** – If `-1`, no dropout will be applied. Otherwise a number between 0 and 1 is expected, denoting the dropout rate of hidden layers.
- **use_spectral_norm** – Use spectral normalization for training.
- **use_batch_norm** (*bool*) – Whether batch normalization should be used. Will be applied before the activation function in all hidden layers.
- **bn_track_stats** (*bool*) – If batch normalization is used, then this option determines whether running statistics are tracked in these layers or not (see argument `track_running_stats` of class `utils.batchnorm_layer.BatchNormLayer`).

  If `False`, then batch statistics are utilized even during evaluation. If `True`, then running stats are tracked. When using this network in a continual learning scenario with different tasks then the running statistics are expected to be maintained externally. The argument `stats_id` of the method `utils.batchnorm_layer.BatchNormLayer.forward()` can be provided using the argument `condition` of method *forward()*.

  ### Example

  To maintain the running stats, one can simply iterate over all batch norm layers and checkpoint the current running stats (e.g., after learning a task when applying a Continual learning scenario).

  ```
  for bn_layer in net.batchnorm_layers:
      bn_layer.checkpoint_stats()
  ```

- **distill_bn_stats** (*bool*) – If `True`, then the shapes of the batchnorm statistics will be added to the attribute `mnets.mnet_interface.MainNetInterface.hyper_shapes_distilled` and the current statistics will be returned by the method *distillation_targets()*.

  Note, this attribute may only be `True` if `bn_track_stats` is `True`.
- **use_context_mod** (*bool*) – Add context-dependent modulation layers `utils.context_mod_layer.ContextModLayer` after the linear computation of each layer.

- **context_mod_inputs** (*bool*) – Whether context-dependent modulation should also be applied to network intpus directly. I.e., assume $\mathbf{x}$ is the input to the network. Then the first network operation would be to modify the input via $\mathbf{x} \cdot \mathbf{g} + \mathbf{s}$ using context- dependent gain and shift parameters.

  Note: Argument applies only if `use_context_mod` is `True`.

- **no_last_layer_context_mod** (*bool*) – If `True`, context-dependent modulation will not be applied to the output layer.

  Note: Argument applies only if `use_context_mod` is `True`.

- **context_mod_no_weights** (*bool*) – The weights of the context-mod layers (`utils.context_mod_layer.ContextModLayer`) are treated independently of the option `no_weights`. This argument can be used to decide whether the context-mod parameters (gains and shifts) are maintained internally or externally.

  Note: Check out argument `weights` of the *forward()* method on how to correctly pass weights to the network that are externally maintained.

- **context_mod_post_activation** (*bool*) – Apply context-mod layers after the activation function (`activation_fn`) in hidden layer rather than before, which is the default behavior.

  Note: This option only applies if `use_context_mod` is `True`.

  Note: This option does not affect argument `context_mod_inputs`.

  Note: This option does not affect argument `no_last_layer_context_mod`. Hence, if a output-nonlinearity is applied through argument `out_fn`, then context-modulation would be applied before this non-linearity.

- **context_mod_gain_offset** (*bool*) – Activates option `apply_gain_offset` of class `utils.context_mod_layer.ContextModLayer` for all context-mod layers that will be instantiated.
- **context_mod_gain_softplus** (*bool*) – Activates option `apply_gain_softplus` of class `utils.context_mod_layer.ContextModLayer` for all context-mod layers that will be instantiated.
- **out_fn** (*optional*) – If provided, this function will be applied to the output neurons of the network.

  Warning: This changes the interpretation of the output of the *forward()* method.

- **verbose** (*bool*) – Whether to print information (e.g., the number of weights) during the construction of the network.

Initializes internal Module state, shared by both nn.Module and ScriptModule.

distillation_targets()

Targets to be distilled after training.

See docstring of abstract super method mnets.mnet_interface.MainNetInterface.distillation_targets().

This method will return the current batch statistics of all batch normalization layers if distill_bn_stats and use_batch_norm was set to True in the constructor.

Returns

The target tensors corresponding to the shapes specified in attribute hyper_shapes_distilled.

forward(*x*, *weights=None*, *distilled_params=None*, *condition=None*)

Compute the output y of this network given the input x.

Parameters

- **(....)** – See docstring of method mnets.mnet_interface.MainNetInterface.forward(). We provide some more specific information below.
- **weights** (*list or dict*) – If a list of parameter tensors is given and context modulation is used (see argument use_context_mod in constructor), then these parameters are interpreted as context- modulation parameters if the length of weights equals 2*len(net.context_mod_layers). Otherwise, the length is expected to be equal to the length of the attribute mnets.mnet_interface.MainNetInterface.param_shapes.

  Alternatively, a dictionary can be passed with the possible keywords internal_weights and mod_weights. Each keyword is expected to map onto a list of tensors. The keyword internal_weights refers to all weights of this network except for the weights of the context-modulation layers. The keyword mod_weights, on the other hand, refers specifically to the weights of the context-modulation layers. It is not necessary to specify both keywords.
- **distilled_params** – Will be passed as running_mean and running_var arguments of method utils.batchnorm_layer.BatchNormLayer.forward() if batch normalization is used.
- **condition** (*int or dict, optional*) – If int is provided, then this argument will be passed as argument stats_id to the method utils.batchnorm_layer.BatchNormLayer.forward() if batch normalization is used.

  If a dict is provided instead, the following keywords are allowed:

  – bn_stats_id: Will be handled as stats_id of the batchnorm layers as described above.

  – cmod_ckpt_id: Will be passed as argument ckpt_id to the method utils.context_mod_layer.ContextModLayer.forward().

Returns

Tuple containing:

- **y**: The output of the network.
- **h_y** (optional): If out_fn was specified in the constructor, then this value will be returned. It is the last hidden activation (before the out_fn has been applied).

Return type

(tuple)

training: bool

static weight_shapes(*n_in=1*, *n_out=1*, *hidden_layers=[10, 10]*, *use_bias=True*)

Compute the tensor shapes of all parameters in a fully-connected network.

Parameters

- **n_in** – Number of inputs.
- **n_out** – Number of output units.
- **hidden_layers** – A list of ints, each number denoting the size of a hidden layer.
- **use_bias** – Whether the FC layers should have biases.

Returns

A list of list of integers, denoting the shapes of the individual parameter tensors.

4.6 Main-Network Interface

The module `mnets.mnet_interface` contains an interface for main networks. The interface ensures that we can consistently use these networks without knowing their specific implementation.

class hypnettorch.mnets.mnet_interface.**MainNetInterface**

Bases: ABC

A general interface for main networks, that can be used stand-alone (i.e., having their own weights) or with no (or only some) internal weights, such that the remaining weights have to be passed through the forward function (e.g., they may be generated through a hypernetwork).

property batchnorm_layers

A list of instances of class `utils.batchnorm_layer.BatchNormLayer` in case batch normalization is used in this network.

Note: We explicitly do not support the usage of PyTorch its batchnorm layers as class `utils.batchnorm_layer.BatchNormLayer` represents a hypernet compatible wrapper for them.

Type

torch.nn.ModuleList

property context_mod_layers

A list of instances of class `utils.context_mod_layer.ContextModLayer` in case these are used in this network.

Type

torch.nn.ModuleList

custom_init(*normal_init=False*, *normal_std=0.02*, *zero_bias=True*)

Initialize weight tensors in attribute *layer_weight_tensors* using Xavier initialization and set bias vectors to 0.

Note: This method will override the default initialization of the network, which is often based on `torch.nn.init.kaiming_uniform_()` for weight tensors (i.e., attribute *layer_weight_tensors*) and a uniform init based on fan-in/fan-out for bias vectors (i.e., attribute *layer_bias_vectors*).

Parameters

- **normal_init** (*bool*) – Use normal initialization rather than Xavier.
- **normal_std** (*float*) – The standard deviation when choosing `normal_init`.
- **zero_bias** (*bool*) – Whether bias vectors should be initialized to zero. If `False`, then bias vectors are left untouched.

abstract distillation_targets()

Targets to be distilled after training.

If *hyper_shapes_distilled* is not `None`, then this method can be used to retrieve the targets that should be distilled into an external hypernetwork after training.

The shapes of the returned tensors have to match the shapes specified in *hyper_shapes_distilled*.

Example

Assume a continual learning scenario with a main network that uses batch normalization (and tracks running statistics). Then this method should be called right after training on a task in order to retrieve the running statistics, such that they can be distilled into a hypernetwork.

Returns

The target tensors corresponding to the shapes specified in attribute *hyper_shapes_distilled*.

static flatten_params(*params*, *param_shapes=None*, *unflatten=False*)

Flatten a list of parameter tensors.

This function will take a list of parameter tensors and flatten them into a single vector. This flattening operation can also be undone using the argument `unflatten`.

Parameters

- **params** (*list*) – A list of tensors. Those tensors will be flattened and concatenated into a tensor. If `unflatten=True`, then `params` is expected to be a flattened tensor, which will be split into a list of tensors according to `param_shapes`.
- **param_shapes** (*list*) – List of parameter tensor shapes. Required when unflattening a flattened parameter tensor.
- **unflatten** (*bool*) – If `True`. the flattening operation will be reversed.

Returns

The flattened tensor. If `unflatten=True`, a list of tensors will be returned.

Return type

(torch.Tensor)

abstract forward(*x*, *weights=None*, *distilled_params=None*, *condition=None*)

Compute the output y of this network given the input x.

Parameters

- **x** – The inputs x to the network.
- **weights** (*optional*) – List of weight tensors, that are used as network parameters. If attribute *hyper_shapes_learned* is not `None`, then this argument is non-optional and the shapes of the weight tensors have to be as specified by *hyper_shapes_learned*.

Otherwise, this option might still be set but the weight tensors must follow the shapes specified by attribute *param_shapes*.

- **distilled_params** (*optional*) – May only be passed if attribute *hyper_shapes_distilled* is not `None`.

  If not passed but the network relies on those parameters (e.g., batchnorm running statistics), then this method simply chooses the current internal representation of these parameters as returned by *distillation_targets()*.

- **condition** (*optional*) – Sometimes, the network will have to be conditioned on contextual information, which can be passed via this argument and depends on the actual implementation of this interface.

  For instance, when using batch normalization in a continual learning scenario, where running statistics have been checkpointed for every task, then this `condition` might be the actual task ID, that is passed as the argument `stats_id` of the method `utils.batchnorm_layer.BatchNormLayer.forward()`.

Returns
The output y of the network.

get_output_weight_mask(*out_inds=None*, *device=None*)

Create a mask for selecting weights connected solely to certain output units.

This method will return a list of the same length as *param_shapes*. Entries in this list are either `None` or masks for the corresponding parameter tensors. For all parameter tensors that are not directly connected to output units, the corresponding entry will be `None`. If `out_inds is None`, then all output weights are selected by a masking value `1`. Otherwise, only the weights connected to the output units in `out_inds` are selected, the rest is masked out.

Note: This method only works for networks with a fully-connected output layer (see *has_fc_out*), that have the attribute *mask_fc_out* set. Otherwise, the method has to be overwritten by an implementing class.

Parameters

- **out_inds** (*list, optional*) – List of integers. Each entry denotes an output unit.
- **device** – Pytorch device. If given, the created masks will be moved onto this device.

Returns
List of masks with the same length as *param_shapes*. Entries whose corresponding parameter tensors are not connected to the network outputs are `None`.

Return type
(list)

property has_bias

Whether layers in this network have bias terms.

Type
bool

property has_fc_out

Whether the output layer of the network is a fully-connected layer.

Type
bool

property has_linear_out

Is `True` if no nonlinearity is applied in the output layer.

Type
bool

property hyper_shapes_distilled

A list of lists of integers. This attribute is complementary to attribute *hyper_shapes_learned*, which contains shapes of tensors that are learned through the hypernetwork. In contrast, this attribute should contain the shapes of tensors that are not needed by the main network during training (as it learns or calculates the tensors itself), but should be distilled into a hypernetwork after training in order to avoid increasing memory consumption.

The attribute is `None` if no tensors have to be distilled into a hypernetwork.

For instance, if batch normalization is used, then the attribute *hyper_shapes_learned* might contain the batch norm weights whereas the attribute *hyper_shapes_distilled* contains the running statistics, which are first estimated by the main network during training and later distilled into the hypernetwork.

Type
list or None

property hyper_shapes_learned

A list of lists of integers. Each list represents the shape of a weight tensor that has to be passed to the *forward()* method during training. If all weights are maintained internally, then this attribute will be `None`.

Type
list

property hyper_shapes_learned_ref

A list of integers. Each entry either represents an index within attribute *param_shapes* or is set to `-1`.

Note: The possibility that entries may be `-1` should account for unforeseeable flexibility that programmers may need.

Type
list

property internal_params

A list of all internally maintained parameters of the main network currently in use. If all parameters are assumed to be generated externally, then this attribute will be `None`.

Simply speaking, the parameters listed here should be passed to the optimizer.

Note: In most cases, the attribute will contain the same set of parameter objects as the method `torch.nn.Module.parameters()` would return. Though, there might be future use-cases where the programmer wants to hide parameters from the optimizer in a task- or time-dependent manner.

Type
torch.nn.ParameterList or None

property internal_params_ref

A list of integers. Each entry either represents an index within attribute *param_shapes* or is set to -1.

Can only be spacified if *internal_params* is not None.

Note: The possibility that entries may be -1 should account for unforeseeable flexibility that programmers may need.

Type
list or None

property layer_bias_vectors

Similar to attribute *layer_weight_tensors* but for the bias vectors in each layer. List should be empty in case *has_bias* is False.

Note: There might be cases where some weight matrices in attribute *layer_weight_tensors* have no bias vectors, in which case elements of this list might be None.

Type
torch.nn.ParameterList

property layer_weight_tensors

These are the actual weight tensors used in layers (e.g., weight matrix in fully-connected layer, kernels in convolutional layer, ...).

This attribute is useful when applying a custom initialization to these layers.

Type
torch.nn.ParameterList

property mask_fc_out

If this attribute is set to True, it is implicitly assumed that if *hyper_shapes_learned* is not None, the last two entries of *hyper_shapes_learned* are the weights and biases of the final fully-connected layer.

This attribute is helpful, for instance, in multi-head continual learning settings. In case we regularize task-specific main network weights, it is important to know which weights are specific for an output head (as determined by the weights of the final layer).

Note: Only applies if attribute *has_fc_out* is True.

Type
bool

property num_internal_params

The number of internally maintained parameters as prescribed by attribute *internal_params*.

Type
int

property num_params

The total number of weights in the parameter tensors described by the attribute *param_shapes*.

Type
int

overwrite_internal_params(*new_params*)

Overwrite the values of all internal parameters.

This will affect all parameters maintained in attribute *internal_params*.

An example usage of this method is the initialization of a standalone main network with weights that have been previously produced by a hypernetwork.

Parameters
new_params – A list of parameter values that are used to initialize the network internal parameters is expected.

property param_shapes

A list of lists of integers. Each list represents the shape of a parameter tensor. Note, this attribute is independent of the attribute *internal_params*, it always comprises the shapes of all parameter tensors as if the network would be stand-alone (i.e., no weights being passed to the *forward()* method).

Type
list

property param_shapes_meta

A list of dictionaries. The length of the list is equal to the length of the list *param_shapes* and each entry of this list provides meta information to the corresponding entry in *param_shapes*. Each dictionary contains the keys `name`, `index` and `layer`. The key `name` is a string and refers to the type of weight tensor that the shape corresponds to.

Possible values are:

- `'weight'`: A weight tensor of a standard layer as those stored in attribute *layer_weight_tensors*.
- `'bias'`: A bias vector of a standard layer as those stored in attribute *layer_bias_vectors*.
- `'bn_scale'`: The weights for scaling activations in a batchnorm layer `utils.batchnorm_layer.BatchNormLayer`.
- `'bn_shift'`: The weights for shifting activations in a batchnorm layer `utils.batchnorm_layer.BatchNormLayer`.
- `'cm_scale'`: The weights for scaling activations in a context-mod layer `utils.context_mod_layer.ContextModLayer`.
- `'cm_shift'`: The weights for shifting activations in a context-mod layer `utils.context_mod_layer.ContextModLayer`.
- `'embedding'`: The parameters represent embeddings.
- `None`: Not specified!

The key `index` might refer to the index of the corresponding parameter tensor (if existing) inside the *internal_params* list. It is `-1` if the parameter tensor is not internally maintained.

The key `layer` is an integer. Shapes with the same `layer` entry are supposed to reside in the same layer. For instance, a `'weight'` and a `'bias'` with the same entry for key `layer` are supposed to be the weight tensor and bias vector in the same layer. The value `-1` refers to *not specified*!

type
list

static shapes_to_num_weights(*dims*)

The number of parameters contained in a list of tensors with the given shapes.

Parameters
dims – List of tensor shapes. For instance, the attribute *hyper_shapes_learned*.

Returns
(int)

property weights

Same as *internal_params*.

Deprecated since version 1.0: Please use attribute *internal_params* instead.

Type
torch.nn.ParameterList or None

4.7 ResNet

This module implements the class of Resnet networks described in section 4.2 of the following paper:

"Deep Residual Learning for Image Recognition", He et al., 2015 https://arxiv.org/abs/1512.03385

class hypnettorch.mnets.resnet.**ResNet**(*in_shape=(32, 32, 3)*, *num_classes=10*, *use_bias=True*, *num_feature_maps=(16, 16, 32, 64)*, *verbose=True*, *n=5*, *k=1*, *no_weights=False*, *init_weights=None*, *use_batch_norm=True*, *bn_track_stats=True*, *distill_bn_stats=False*, *context_mod_apply_pixel_wise=False*, ***kwargs*)

Bases: *Classifier*

A resnet with $6n + 2$ layers with $3n$ residual blocks, consisting of two layers each.

Parameters

- **in_shape** (*tuple or list*) – The shape of an input sample in format HWC.

  Note
  We assume the Tensorflow format, where the last entry denotes the number of channels.

- **num_classes** (*int*) – The number of output neurons.

  Note: The network outputs logits.

- **use_bias** (*bool*) – Whether layers may have bias terms.

  Note: Bias terms are unnecessary in convolutional layers if batch normalization is used. However, this option disables bias terms altogether (including in the final fully-connected layer).

- **num_feature_maps** (*tuple*) – A list of 4 integers, each denoting the number of feature maps of convolutional layers in a certain group of the network architecture. The first entry is the number of feature maps of the first convolutional layer, the remaining 3 numbers determine the number of feature maps in the consecutive groups comprising $2n$ convolutional layers each.

- **verbose** (*bool*) – Allow printing of general information about the generated network (such as number of weights).
- **n** (*int*) – The network will consist of $6n + 2$ layers. In the paper n has been chosen to be 3, 5, 7, 9 or 18.
- **k** (*int*) – The widening factor. Feature maps in the 3 convolutional groups will be multiplied by this number. See argument `num_feature_maps`. This argument is typical for wide resnets, such as `mnets.wide_resnet.WRN`. Hence, for `k > 1` this network becomes essentially a wide resnet.
- **no_weights** (*bool*) – If set to `True`, no trainable parameters will be constructed, i.e., weights are assumed to be produced ad-hoc by a hypernetwork and passed to the *forward()* method.

  Note, this also affects the affine parameters of the batchnorm layer. I.e., if set to `True`, then the argument `affine` of `utils.batchnorm_layer.BatchNormLayer` will be set to `False` and we expect the batchnorm parameters to be passed to the *forward()*.
- **init_weights** (*optional*) – This option is for convinience reasons. The option expects a list of parameter values that are used to initialize the network weights. As such, it provides a convinient way of initializing a network with a weight draw produced by the hypernetwork.
- **use_batch_norm** – Whether batch normalization should used. It will be applied after all convolutional layers (before the activation).
- **bn_track_stats** – If batch normalization is used, then this option determines whether running statistics are tracked in these layers or not (see argument `track_running_stats` of class `utils.batchnorm_layer.BatchNormLayer`).

  If `False`, then batch statistics are utilized even during evaluation. If `True`, then running stats are tracked. When using this network in a continual learning scenario with different tasks then the running statistics are expected to be maintained externally. The argument `stats_id` of the method `utils.batchnorm_layer.BatchNormLayer.forward()` can be provided using the argument `condition` of method *forward()*.

  ### Example

  To maintain the running stats, one can simply iterate over all batch norm layers and checkpoint the current running stats (e.g., after learning a task when applying a Continual Learning scenario).

  ```
  for bn_layer in net.batchnorm_layers:
      bn_layer.checkpoint_stats()
  ```

- **distill_bn_stats** – If `True`, then the shapes of the batchnorm statistics will be added to the attribute `mnets.mnet_interface.MainNetInterface.hyper_shapes_distilled` and the current statistics will be returned by the method *distillation_targets()*.

  Note, this attribute may only be `True` if `bn_track_stats` is `True`.
- **context_mod_apply_pixel_wise** (*bool*) – By default, the context-dependent modulation applies a scalar gain and shift to all feature maps in the output of a convolutional layer. When activating this option, the gain and shift will be a per-pixel parameter in all feature maps.

  To be more precise, consider the output of a convolutional layer of shape `[C,H,W]`. By default, there will be `C` gain and shift parameters for such a layer. Upon activating this option, the number of gain and shift parameters for such a layer will increase to `C x H x W`.

- ****kwargs** – Keyword arguments regarding context modulation. This class can process the same context-modulation related arguments as class `mnets.mlp.MLP`. Additionally, one may specify the argument `context_mod_apply_pixel_wise`.

  Some additional remarks regarding the handling of keyword arguments:

  – `use_context_mod`: Context-modulation will be applied after the linear computation of each layer (i.e. all hidden layers (conv layers) as well as the final FC output layer).

  Similar to Spatial Batch-Normalization, there will be a scalar shift and gain applied per feature map for all convolutional layers (except if `context_mod_apply_pixel_wise` is set).

  – `context_mod_inputs`: The input is treated like the output of a convolutional layer when applying context-dependent modulation.

Initialize the network.

Parameters

- **num_classes** – The number of output neurons.
- **verbose** – Allow printing of general information about the generated network (such as number of weights).

`distillation_targets`()

Targets to be distilled after training.

See docstring of abstract super method `mnets.mnet_interface.MainNetInterface.distillation_targets()`.

This method will return the current batch statistics of all batch normalization layers if `distill_bn_stats` and `use_batch_norm` were set to `True` in the constructor.

Returns

The target tensors corresponding to the shapes specified in attribute `hyper_shapes_distilled`.

`forward`(*x*, *weights=None*, *distilled_params=None*, *condition=None*)

Compute the output y of this network given the input x.

Parameters

- **(....)** – See docstring of method `mnets.mnet_interface.MainNetInterface.forward()`. We provide some more specific information below.
- **x** (*torch.Tensor*) – Batch of flattened input images.

  Note: We assume the Tensorflow format, where the last entry denotes the number of channels.

- **weights** (*list or dict*) – If a list of parameter tensors is given and context modulation is used (see argument `use_context_mod` in constructor), then these parameters are interpreted as context- modulation parameters if the length of `weights` equals `2*len(net.context_mod_layers)`. Otherwise, the length is expected to be equal to the length of the attribute `mnets.mnet_interface.MainNetInterface.param_shapes`.

  Alternatively, a dictionary can be passed with the possible keywords `internal_weights` and `mod_weights`. Each keyword is expected to map onto a list of tensors. The keyword `internal_weights` refers to all weights of this network except for the weights of the

context-modulation layers. The keyword `mod_weights`, on the other hand, refers specifically to the weights of the context-modulation layers. It is not necessary to specify both keywords.

- **distilled_params** – Will be passed as `running_mean` and `running_var` arguments of method `utils.batchnorm_layer.BatchNormLayer.forward()` if batch normalization is used.
- **condition** (*optional*, *int* or *dict*) – If `int` is provided, then this argument will be passed as argument `stats_id` to the method `utils.batchnorm_layer.BatchNormLayer.forward()` if batch normalization is used.

  If a `dict` is provided instead, the following keywords are allowed:

  - `bn_stats_id`: Will be handled as `stats_id` of the batchnorm layers as described above.
  - `cmod_ckpt_id`: Will be passed as argument `ckpt_id` to the method `utils.context_mod_layer.ContextModLayer.forward()`.

Returns
The output of the network.

Return type
(torch.Tensor)

training: bool

4.8 ResNet for ImageNet

This module implements the class of Resnet networks described Table 1 of the following paper:

"Deep Residual Learning for Image Recognition", He et al., 2015 https://arxiv.org/abs/1512.03385

Those networks are designed for inputs of size 224 x 224. In contrast, the Resnet family implemented by class `mnets.resnet.ResNet` is primarily designed for CIFAR like inputs of size 32 x 32.

class hypnettorch.mnets.resnet_imgnet.**ResNetIN**(*in_shape=(224, 224, 3)*, *num_classes=1000*, *use_bias=True*, *use_fc_bias=None*, *num_feature_maps=(64, 64, 128, 256, 512)*, *blocks_per_group=(2, 2, 2, 2)*, *projection_shortcut=False*, *bottleneck_blocks=False*, *cutout_mod=False*, *no_weights=False*, *use_batch_norm=True*, *bn_track_stats=True*, *distill_bn_stats=False*, *chw_input_format=False*, *verbose=True*, ***kwargs*)

Bases: *Classifier*

Hypernet-compatible Resnets for ImageNet.

The architecture of those Resnets is summarized in Table 4 of He et al.. They consist of 5 groups of convolutional layers, where the first group only has 1 convolutional layer followed by a max-pooling operation. The other 4 groups consist of blocks (see `blocks_per_group`) of either 2 or 3 (see `bottleneck_blocks`) convolutional layers per block. The network then computes its output via a final average pooling operation and a fully- connected layer.

The number of layer per network is therewith `1 + sum(blocks_per_group) * 2 + 1`, i.e., initial conv layer, num. conv layers in all blocks (assuming `bottleneck_blocks=False`) and the final fully-connected layer. If

`projection_shortcut=True`, additional 1x1 conv layers are added for shortcuts where the feature maps tensor shape changes.

Here are a few implementation details worth noting:

- If `use_batch_norm=True`, it would be redundant to add convolutional layers to the conv layers, therefore one should set `use_bias=False, use_fc_bias=True`. Skip connections never use biases.
- Online implementations vary in their use of projection or identity shortcuts. We offer both possibilities (`projection_shortcut`). If `projection_shortcut` is used, then a batchnorm layer is added after each projection.

Here are parameter configurations that can be used to obtain well-known Resnets (all configurations should use `use_bias=False, use_fc_bias=True`):

- *Resnet-18*: `blocks_per_group=(2,2,2,2), bottleneck_blocks=False`
- *Resnet-34*: `blocks_per_group=(3,4,6,3), bottleneck_blocks=False`
- *Resnet-50*: `blocks_per_group=(3,4,6,3), bottleneck_blocks=True`
- *Resnet-101*: `blocks_per_group=(3,4,23,3), bottleneck_blocks=True`
- *Resnet-152*: `blocks_per_group=(3,4,36,3), bottleneck_blocks=True`

Parameters

- **`(....)`** – See arguments of class:*mnets.wrn.WRN*.
- **`num_feature_maps`** (*tuple*) – A list of 5 integers, each denoting the number of feature maps in a group of convolutional layers.

  Note: If `bottleneck_blocks=True`, then the last 1x1 conv layer in each block has 4 times as many feature maps as specified by this argument.

- **`blocks_per_group`** (*tuple*) – A list of 4 integers, each denoting the number of convolutional blocks for the groups of convolutional layers.
- **`projection_shortcut`** (*bool*) – If `True`, skip connections that otherwise would require zero-padding or subsampling will be realized via 1x1 conv layers followed by batchnorm. All other skip connections will be realized via identity mappings.
- **`bottleneck_blocks`** (*bool*) – Whether normal blocks or bottleneck blocks should be used (cf. Fig. 5 in He et al.)
- **`cutout_mod`** (*bool*) – Sometimes, networks from this family are used for smaller (CIFAR-like) images. In this case, one has to either upscale the images or adapt the architecture slightly (otherwise, small images are too agressively downscaled at the very beginning).

  When activating this option, the first conv layer is modified as described here, i.e., it uses a kernel size of 3 with stride 1 and the max-pooling layer is omitted.

  Note, in order to recover the same architecture as in the link above one has to additionally set: `use_bias=False, use_fc_bias=True, projection_shortcut=True`.

Initialize the network.

Parameters

- **`num_classes`** – The number of output neurons.

- **verbose** – Allow printing of general information about the generated network (such as number of weights).

distillation_targets()

Targets to be distilled after training.

See docstring of abstract super method mnets.mnet_interface.MainNetInterface.distillation_targets().

This method will return the current batch statistics of all batch normalization layers if distill_bn_stats and use_batch_norm were set to True in the constructor.

Returns
The target tensors corresponding to the shapes specified in attribute hyper_shapes_distilled.

forward(*x*, *weights=None*, *distilled_params=None*, *condition=None*)

Compute the output y of this network given the input x.

Parameters

- **(....)** – See docstring of method mnets.resnet.ResNet.forward(). We provide some more specific information below.
- **x** (*torch.Tensor*) – Based on the constructor argument chw_input_format, either a flattened image batch with encoding HWC or an unflattened image batch with encoding CHW is expected.

Returns
The output of the network.

Return type
(torch.Tensor)

get_output_weight_mask(*out_inds=None*, *device=None*)

Create a mask for selecting weights connected solely to certain output units.

See docstring of overwritten super method mnets.mnet_interface.MainNetInterface.get_output_weight_mask().

property has_bias

Getter for read-only attribute *has_bias*.

training: bool

4.9 SimpleRNN

Implementation of a simple recurrent neural network that has stacked vanilla RNN or LSTM layers that are optionally enclosed by fully-connected layers.

An example usage is as a main model, where the main weights are initialized and protected by a method such as EWC, and the context-modulation patterns of the neurons are produced by an external hypernetwork.

class hypnettorch.mnets.simple_rnn.**SimpleRNN**(*n_in=1*, *rnn_layers=(10,)*, *fc_layers_pre=()*, *fc_layers=(1,)*, *activation=Tanh()*, *use_lstm=False*, *use_bias=True*, *no_weights=False*, *init_weights=None*, *kaiming_rnn_init=False*, *context_mod_last_step=False*, *context_mod_num_ts=-1*, *context_mod_separate_layers_per_ts=False*, *verbose=True*, ***kwargs*)

Bases: Module, *MainNetInterface*

Implementation of a simple RNN.

This is a simple recurrent network, that receives input vector $\mathbf{x}$ and outputs a vector $\mathbf{y}$ of real values.

Note: The output is non-linear if the last layer is recurrent! Otherwise, logits are returned (cmp. attribute `mnets.mnet_interface.MainNetInterface.has_fc_out`).

Parameters

- **n_in** (*int*) – Number of inputs.
- **rnn_layers** (*list or tuple*) – List of integers. Each entry denotes the size of a recurrent layer. Recurrent layers will simply be stacked as layers of this network.

  If `fc_layers_pre` is empty, then the recurrent layers are the initial layers. If `fc_layers` is empty, then the last entry of this list will denote the output size.

  Note: This list may never be empty.

- **fc_layers_pre** (*list or tuple*) – List of integers. Before the recurrent layers a set of fully-connected layers may be added. This might be specially useful when constructing recurrent autoencoders. The entries of this list will denote the sizes of those layers.

  If `fc_layers_pre` is not empty, its first entry will denote the input size of this network.
- **fc_layers** (*list or tuple*) – List of integers. After the recurrent layers, a set of fully-connected layers is added. The entries of this list will denote the sizes of those layers.

  If `fc_layers` is not empty, its last entry will denote the output size of this network.
- **activation** – The nonlinearity used in hidden layers.
- **use_lstm** (*bool*) – If set to *True* `, the recurrent layers will be LSTM layers.
- **use_bias** (*bool*) – Whether layers may have bias terms.
- **no_weights** (*bool*) – If set to `True`, no trainable parameters will be constructed, i.e., weights are assumed to be produced ad-hoc by a hypernetwork and passed to the *forward()* method.
- **init_weights** (*list, optional*) – This option is for convinience reasons. The option expects a list of parameter values that are used to initialize the network weights. As such, it provides a convinient way of initializing a network with a weight draw produced by the hypernetwork.

  Note, internal weights (see `mnets.mnet_interface.MainNetInterface.weights`) will be affected by this argument only.
- **kaiming_rnn_init** (*bool*) – By default, PyTorch initializes its recurrent layers uniformly with an interval defined by the square-root of the inverse of the layer size.

  If this option is enabled, then the recurrent layers will be initialized using the kaiming init as implemented by the function `utils.torch_utils.init_params()`.
- **context_mod_last_step** (*bool*) – Whether context modulation is applied at the last time step os a recurrent layer only. If `False`, context modulation is applied at every time step.

Note: This option only applies if `use_context_mod` is `True`.

- **context_mod_num_ts** (*int, optional*) – The maximum number of timesteps. If specified, context-modulation with a different set of weights is applied at every timestep. If `context_mod_separate_layers_per_ts` is `True`, then a separate context-mod layer per timestep will be created. Otherwise, a single context-mod layer is created, but the expected parameter shapes for this layer are `[context_mod_num_ts, *context_mod_shape]`.

  Note: This option only applies if `use_context_mod` is `True`.

- **context_mod_separate_layers_per_ts** (*bool*) – If specified, a separate context-mod layer per timestep is created (required if `context_mod_no_weights` is `False`).

  Note: Only applies if `context_mod_num_ts` is specified.

- **verbose** (*bool*) – Whether to print information (e.g., the number of weights) during the construction of the network.
- ****kwargs** – Keyword arguments regarding context modulation. This class can process the same context-modulation related arguments as class `mnets.mlp.MLP` (plus the additional ones noted above).

Initializes internal Module state, shared by both nn.Module and ScriptModule.

basic_rnn_step(*d*, *t*, *x_t*, *h_t*, *int_weights*, *cm_weights*, *ckpt_id*, *is_last_step*)

Perform vanilla rnn pass from inputs to hidden units.

Apply context modulation if necessary (i.e. if `cm_weights` is not `None`).

This function implements a step of an Elman RNN.

Note: We made the following design choice regarding context-modulation. In contrast to the LSTM, the Elman network layer consists of "two steps", updating the hidden state and computing an output based on this hidden state. To be fair, context-mod should influence both these "layers". Therefore, we apply context-mod twice, but using the same weights. This of course assumes that the hidden state and output vector have the same dimensionality.

Parameters

- **d** (*int*) – Index of the layer.
- **t** (*int*) – Current timestep.
- **x_t** – Tensor of size `[batch_size, n_hidden_prev]` with inputs.
- **h_t** (*tuple*) – Tuple of length 2, containing two tensors of size `[batch_size, n_hidden]` with previous hidden states `h` and and previous outputs `y`.

  Note: The previous outputs `y` are ignored by this method, since they are not required in an Elman RNN step.

- **int_weights** – See docstring of method *compute_hidden_states()*.

- **cm_weights** (*list*) – The weights of the context-mod layer, if context- mod should be applied.
- **ckpt_id** – See docstring of method *compute_hidden_states()*.
- **is_last_step** (*bool*) – Whether the current time step is the last one.

Returns

Tuple containing:

- **h_t** (torch.Tensor): The tensor `h_t` of size `[batch_size, n_hidden]` with the new hidden state.
- **y_t** (torch.Tensor): The tensor `y_t` of size `[batch_size, n_hidden]` with the new cell state.

Return type
(tuple)

property bptt_depth

The truncation depth for backprop through time.

If `-1`, backprop through time (BPTT) will unroll all timesteps present in the input. Otherwise, the forward pass will detach the RNN hidden states smaller or equal than `num_timesteps - bptt_depth` timesteps, resulting in truncated BPTT (T-BPTT).

Type
int

compute_basic_rnn_output(*h_t*, *int_weights*, *use_cm*, *cm_weights*, *cm_idx*, *ckpt_id*, *is_last_step*)

Compute the output of a vanilla RNN given the hidden state.

Parameters

- **(...)** – See docstring of method *basic_rnn_step()*.
- **use_cm** (*boolean*) – Whether context modulation is being used.
- **cm_idx** (*int*) – Index of the context-mod layer.

Returns
The output.

Return type
(torch.tensor)

compute_fc_outputs(*h*, *fc_w_weights*, *fc_b_weights*, *num_fc_cm_layers*, *cm_fc_layer_weights*, *cm_offset*, *cmod_cond*, *is_post_fc*, *ret_hidden*)

Compute the forward pass through the fully-connected layers.

This method also appends activations to `ret_hidden`.

Parameters

- **h** (*torch.Tensor*) – The input from the previous layer.
- **fc_w_weights** (*list*) – The weights for the fc layers.
- **fc_b_weights** (*list*) – The biases for the fc layers.
- **num_fc_cm_layers** (*int*) – The number of context-modulation layers associated with this set of fully-connected layers.

- **cm_fc_layer_weights** (*list*) – The context-modulation weights associated with the current layers.
- **cm_offset** (*int*) – The index to access the correct context-mod layers.
- **cmod_cond** (*bool*) – Some condition to perform context modulation.
- **is_post_fc** (*bool*) – layers of the network. In this case, there will be no activation applied to the last layer outputs.
- **ret_hidden** (*list or None*) – The list where to append the hidden recurrent activations.

Returns

Tuple containing:

- **ret_hidden**: The hidden recurrent activations.
- **h**: Transformed activation h.

Return type
(Tuple)

compute_hidden_states(*x*, *layer_ind*, *int_weights*, *cm_weights*, *ckpt_id*, *h_0=None*, *c_0=None*)

Compute the hidden states for the recurrent layer `layer_ind` from a sequence of inputs x.

If so specified, context modulation is applied before or after the nonlinearities.

Parameters

- **x** – The inputs x to the layer. x has shape `[sequence_len, batch_size, n_hidden_prev]`.
- **layer_ind** (*int*) – Index of the layer.
- **int_weights** – Internal weights associated with this recurrent layer.
- **cm_weights** – Context modulation weights.
- **ckpt_id** – Will be passed as option `ckpt_id` to method `utils.context_mod_layer.ContextModLayer.forward()` if context-mod layers are used.
- **h_0** (*torch.Tensor, optional*) – The initial state for h.
- **c_0** (*torch.Tensor, optional*) – The initial state for c. Note that for LSTMs, if the initial state is to be defined, this variable is necessary also, not only h_0, whereas for vanilla RNNs it is enough to provide h_0 as c_0 represents the output of the layer and it can be easily computed from *h_0*.

Returns

Tuple containing:

- **outputs** (torch.Tensor): The sequence of visible hidden states given the input. It has shape `[sequence_len, batch_size, n_hidden]`.
- **hiddens** (torch.Tensor): The sequence of hidden states given the input. For LSTMs, this corresponds to c. It has shape `[sequence_len, batch_size, n_hidden]`.

Return type
(tuple)

distillation_targets()

Targets to be distilled after training.

See docstring of abstract super method mnets.mnet_interface.MainNetInterface.distillation_targets().

This network does not have any distillation targets.

Returns
None

forward(*x*, *weights=None*, *distilled_params=None*, *condition=None*, *return_hidden=False*, *return_hidden_int=False*)

Compute the output y of this network given the input x.

Parameters

- **(....)** – See docstring of method mnets.mnet_interface.MainNetInterface.forward(). We provide some more specific information below.
- **weights** (*list or dict*) – See argument weights of method mnets.mlp.MLP.forward().
- **condition** (*optional, int*) – If provided, then this argument will be passed as argument ckpt_id to the method utils.context_mod_layer.ContextModLayer.forward().
- **return_hidden** (*bool, optional*) – If True, all hidden activations of fully-connected and recurrent layers (where we defined y_t as hidden state of vannila RNN layers as these are the layer output passed to the next layer) are returned.

  Specifically, hidden activations are the outputs of each hidden layer that are passed to the next layer.
- **return_hidden_int** (*bool, optional*) – If True, in addition to hidden, an additional variable hidden_int is returned containing the internal hidden states of recurrent layers (i.e., the cell states c_t for LSTMs and the actual hidden state h_t for Elman layers) are returned. Since fully- connected layers have no such internal hidden activations, the corresponding entry in hidden_int will be None.

Returns

Where the tuple is containing:

- **output** (torch.Tensor): The output of the network.
- **hidden** (list): If return_hidden is True, then the hidden activities of the layers are returned, which have the shape (seq_length, batch_size, n_hidden).
- **hidden_int** (list): If return_hidden_int is True, then in addition to hidden a tensor hidden_int per recurrent layer is returned containing internal hidden states. The list will contain a None entry for each fully-connected layer to ensure same length as hidden.

Return type
(torch.Tensor or tuple)

get_cm_inds()

Get the indices of mnets.mnet_interface.MainNetInterface.param_shapes that are associated with context-modulation.

Returns
List of integers representing indices of mnets.mnet_interface.MainNetInterface.param_shapes.

Return type
(list)

`get_cm_weights`()

Get internal maintained weights that are associated with context- modulation.

Returns
List of weights from `mnets.mnet_interface.MainNetInterface.internal_params` that are belonging to context-mod layers.

Return type
(list)

`get_non_cm_weights`()

Get internal weights that are not associated with context-modulation.

Returns
List of weights from `mnets.mnet_interface.MainNetInterface.internal_params` that are not belonging to context-mod layers.

Return type
(list)

`get_output_weight_mask`(*out_inds=None*, *device=None*)

Get masks to select output weights.

See docstring of overwritten super method `mnets.mnet_interface.MainNetInterface.get_output_weight_mask()`.

`init_hh_weights_orthogonal`()

Initialize hidden-to-hidden weights orthogonally.

This method will overwrite the hidden-to-hidden weights of recurrent layers.

`lstm_rnn_step`(*d*, *t*, *x_t*, *h_t*, *int_weights*, *cm_weights*, *ckpt_id*, *is_last_step*)

Perform an LSTM pass from inputs to hidden units.

Apply masks to the temporal sequence for computing the loss. Obtained from:

https://mlexplained.com/2019/02/15/building-an-lstm-from-scratch-in-pytorch-lstms-in-depth-part-1/

and:

https://d2l.ai/chapter_recurrent-neural-networks/lstm.html

Parameters

- **d** (*int*) – Index of the layer.
- **t** (*int*) – Current timestep.
- **x_t** – Tensor of size `[batch_size, n_inputs]` with inputs.
- **h_t** (*tuple*) – Tuple of length 2, containing two tensors of size `[batch_size, n_hidden]` with previous hidden states `h` and `c`.
- **int_weights** – See docstring of method *basic_rnn_step()*.
- **cm_weights** – See docstring of method *basic_rnn_step()*.
- **ckpt_id** – See docstring of method *basic_rnn_step()*.
- **is_last_step** (*bool*) – See docstring of method *basic_rnn_step()*.

Returns

Tuple containing:

- **h_t** (torch.Tensor): The tensor `h_t` of size `[batch_size, n_hidden]` with the new hidden state.
- **c_t** (torch.Tensor): The tensor `c_t` of size `[batch_size, n_hidden]` with the new cell state.

Return type
(tuple)

`property num_rec_layers`

Number of recurrent layers in this network (i.e., length of constructor argument `rnn_layers`).

Type
int

`split_cm_weights`(*cm_weights*, *condition*, *num_ts=0*)

Split context-mod weights per context-mod layer.

Parameters

- **cm_weights** (*torch.Tensor*) – All context modulation weights.
- **condition** (*optional, int*) – If provided, then this argument will be passed as argument `ckpt_id` to the method `utils.context_mod_layer.ContextModLayer.forward()`.
- **num_ts** (*int*) – The length of the sequences.

Returns

Where the tuple contains:

- **cm_inputs_weights**: The cm input weights.
- **cm_fc_pre_layer_weights**: The cm pre-recurrent weights.
- **cm_rec_layer_weights**: The cm recurrent weights.
- **cm_fc_layer_weights**: The cm post-recurrent weights.
- **n_cm_rec**: The number of recurrent cm layers.
- **cmod_cond**: The context-mod condition.

Return type
(Tuple)

`split_internal_weights`(*int_weights*)

Split internal weights per layer.

Parameters
int_weights (*torch.Tensor*) – All internal weights.

Returns

Where the tuple contains:

- **fc_pre_w_weights**: The pre-recurrent w weights.
- **fc_pre_b_weights**: The pre-recurrent b weights.
- **rec_weights**: The recurrent weights.

- **fc_w_weights**:The post-recurrent w weights.
- **fc_b_weights**: The post-recurrent b weights.

Return type
(Tuple)

`split_weights`(*weights*)

Split weights into internal and context-mod weights.

Extract which weights should be used, I.e., are we using internally maintained weights or externally given ones or are we even mixing between these groups.

Parameters
weights (*torch.Tensor*) – All weights.

Returns

Where the tuple contains:

- **int_weights**: The internal weights.
- **cm_weights**: The context-mod weights.

Return type
(Tuple)

`training: bool`

`property use_lstm`

See constructor argument `use_lstm`.

Type
bool

4.10 Wide-ResNet

The module `mnets.wide_resnet` implements the class of Wide Residual Networks as described in:

Zagoruyko et al., "Wide Residual Networks", 2017.

`class hypnettorch.mnets.wide_resnet.WRN`(*in_shape=(32, 32, 3)*, *num_classes=10*, *n=4*, *k=10*, *num_feature_maps=(16, 16, 32, 64)*, *use_bias=True*, *use_fc_bias=None*, *no_weights=False*, *use_batch_norm=True*, *bn_track_stats=True*, *distill_bn_stats=False*, *dropout_rate=-1*, *chw_input_format=False*, *verbose=True*, ***kwargs*)

Bases: *Classifier*

Hypernet-compatible Wide Residual Network (WRN).

In the documentation of this class, we follow the notation of the original paper:

- l - deepening factor (number of convolutional layers per residual block). In our case, l is always going to be 2, as this was the configuration found to work best by the authors.
- k - widening factor (multiplicative factor for the number of features in a convolutional layer, see argument `k`).
- $B(3, 3)$ - the block structure. The numbers denote the size of the quadratic kernels used in each convolutional layer from a block. Note, the authors found that $B(3, 3)$ works best, which is why we use this configuration.

- d - total number of convolutional layers. Note, here we deviate from the original notation (where this quantity is called n). Though, we want our notation to stay consistent with the one used in class `mnets.resnet.ResNet`.
- n - number of residual blocks in a group. Note, a resnet consists of 3 groups of residual blocks. See also argument `n` of class `mnets.resnet.ResNet`.

Given this notation, the original paper denotes a WRN architecture via the following notation: *WRN-d-k-B(3,3)*. Note, d contains the total number of convolutional layers (including the input layer and all residual connections that are realized via 1x1 convolutions), but it does not contain the final fully-connected layer. The total depth of the network (assuming residual connection do not add to this depth) remains $6n + 2$ as for `mnets.resnet.ResNet`.

Notable implementation differences to `mnets.resnet.ResNet` (some differences might vanish in the future, this list was updated on 05/06/2020):

- Within a block, convolutional layers are preceeded by a batchnorm layer and the application of the non-linearity. This changes the structure within a block and therefore, residual connections interface with the network at different locations than in class `mnets.resnet.ResNet`.
- Dropout can be used. It will act right after the first convolutional layer of each block.
- If the number of feature maps differs along a skip connection or a downsampling has been applied, 1x1 convolutions rather than padding and manual downsampling is used.

Parameters

- **in_shape** (*tuple or list*) – The shape of an input sample in format `HWC`.

  Note
  We assume the Tensorflow format, where the last entry denotes the number of channels. Also, see argument `chw_input_format`.

- **num_classes** (*int*) – The number of output neurons.

  Note: The network outputs logits.

- **n** (*int*) – The number of residual blocks per group.
- **k** (*int*) – The widening factor. Feature maps in the 3 convolutional groups will be multiplied by this number. See argument `num_feature_maps`.
- **num_feature_maps** (*tuple*) – A list of 4 integers, each denoting the number of feature maps of convolutional layers in a certain group of the network architecture. The first entry is the number of feature maps of the first convolutional layer, the remaining 3 numbers determine the number of feature maps in the consecutive groups comprising $2n$ convolutional layers each.

  Note: The last 3 entries of this list are multiplied by the factor `k`. use_bias (bool): Whether layers may have bias terms.

- **use_bias** (*bool*) – Whether layers may have bias terms.

  Note: Bias terms are unnecessary in convolutional layers if batch normalization is used. However, this option disables bias terms altogether (including in the final fully-connected layer). See option `use_fc_bias`.

- **use_fc_bias** (*optional, bool*) – If None, the value will be linked to use_bias. Otherwise, this option can alter the usage of bias terms in the final layer compared to the remaining (convolutional) layers in the network.
- **no_weights** (*bool*) – If set to True, no trainable parameters will be constructed, i.e., weights are assumed to be produced ad-hoc by a hypernetwork and passed to the *forward()* method.

  Note, this also affects the affine parameters of the batchnorm layer. I.e., if set to True, then the argument affine of utils.batchnorm_layer.BatchNormLayer will be set to False and we expect the batchnorm parameters to be passed to the *forward()*.
- **use_batch_norm** (*bool*) – Whether batch normalization should used. There will be a batchnorm layer after each convolutional layyer (excluding possible 1x1 conv layers in the skip connections). However, the logical order is as follows: batchnorm layer -> ReLU -> convolutional layer. Hence, a residual block (containing multiple of these logical units) starts before a batchnorm layer and ends after a convolutional layer.
- **bn_track_stats** (*bool*) – See argument bn_track_stats of class mnets.resnet.ResNet.
- **distill_bn_stats** (*bool*) – See argument bn_track_stats of class mnets.resnet.ResNet.
- **dropout_rate** (*float*) – If -1, no dropout will be applied. Otherwise a number between 0 and 1 is expected, denoting the dropout rate.

  Dropout will be applied after the first convolutional layers (and before the second batchnorm layer) in each residual block.
- **chw_input_format** (*bool*) – Due to legacy reasons, the network expects by default flattened images as input that were encoded in the HWC format. When enabling this option, the network expects unflattened images in the CHW format (as typical for PyTorch).
- **verbose** (*bool*) – Allow printing of general information about the generated network (such as number of weights).
- ****kwargs** – Keyword arguments regarding context modulation. This class can process the same context-modulation related arguments as class mnets.mlp.MLP. One may additionally specify the argument context_mod_apply_pixel_wise (see class mnets.resnet.ResNet).

Initialize the network.

Parameters

- **num_classes** – The number of output neurons.
- **verbose** – Allow printing of general information about the generated network (such as number of weights).

distillation_targets()

Targets to be distilled after training.

See docstring of abstract super method mnets.mnet_interface.MainNetInterface.distillation_targets().

This method will return the current batch statistics of all batch normalization layers if distill_bn_stats and use_batch_norm were set to True in the constructor.

Returns
The target tensors corresponding to the shapes specified in attribute `hyper_shapes_distilled`.

forward(*x*, *weights=None*, *distilled_params=None*, *condition=None*)

Compute the output y of this network given the input x.

Parameters

- **(....)** – See docstring of method `mnets.resnet.ResNet.forward()`. We provide some more specific information below.
- **x** (*torch.Tensor*) – Based on the constructor argument `chw_input_format`, either a flattened image batch with encoding `HWC` or an unflattened image batch with encoding `CHW` is expected.

Returns
The output of the network.

Return type
(torch.Tensor)

get_output_weight_mask(*out_inds=None*, *device=None*)

Create a mask for selecting weights connected solely to certain output units.

See docstring of overwritten super method `mnets.mnet_interface.MainNetInterface.get_output_weight_mask()`.

property has_bias

Getter for read-only attribute *has_bias*.

training: bool

4.11 The Convnet used by Zenke et al. for CIFAR-10/100

The module `mnets/zenkenet` contains a reimplementation of the network that was used in

"Continual Learning Through Synaptic Intelligence", Zenke et al., 2017. https://arxiv.org/abs/1703.04200

class hypnettorch.mnets.zenkenet.**ZenkeNet**(*in_shape=(32, 32, 3)*, *num_classes=10*, *verbose=True*, *arch='cifar'*, *no_weights=False*, *init_weights=None*, *dropout_rate=0.25*)

Bases: *Classifier*

The network consists of four convolutional layers followed by two fully- connected layers. See implementation for details.

ZenkeNet is a network introduced in

"Continual Learning Through Synaptic Intelligence", Zenke et al., 2017.

See Appendix for details.

We use the same network for a fair comparison to the results reported in the paper.

Parameters

- **in_shape** (*tuple or list*) – The shape of an input sample.

Note: We assume the Tensorflow format, where the last entry denotes the number of channels.

- **num_classes** (*int*) – The number of output neurons. The chosen architecture (see `arch`) will be adopted accordingly.
- **verbose** (*bool*) – Allow printing of general information about the generated network (such as number of weights).
- **arch** (*str*) – The architecture to be employed. The following options are available.
  - `cifar`: The convolutional network used by Zenke et al. for their proposed split CIFAR-10/100 experiment.
- **no_weights** (*bool*) – If set to `True`, no trainable parameters will be constructed, i.e., weights are assumed to be produced ad-hoc by a hypernetwork and passed to the *forward()* method.
- **init_weights** (*optional*) – This option is for convinience reasons. The option expects a list of parameter values that are used to initialize the network weights. As such, it provides a convinient way of initializing a network with a weight draw produced by the hypernetwork.
- **dropout_rate** (*float*) – If `-1`, no dropout will be applied. Otherwise a number between 0 and 1 is expected, denoting the dropout rate.

  Dropout will be applied after the convolutional layers (before pooling) and after the first fully-connected layer (after the activation function).

Note: For the FC layer, the dropout rate is doubled.

Initialize the network.

Parameters

- **num_classes** – The number of output neurons.
- **verbose** – Allow printing of general information about the generated network (such as number of weights).

`distillation_targets()`

Targets to be distilled after training.

See docstring of abstract super method `mnets.mnet_interface.MainNetInterface.distillation_targets()`.

This network does not have any distillation targets.

Returns
`None`

forward(*x*, *weights=None*, *distilled_params=None*, *condition=None*)

Compute the output y of this network given the input x.

Parameters

- **(....)** – See docstring of method `mnets.mnet_interface.MainNetInterface.forward()`. We provide some more specific information below.

- **x** – Input image.

Note: We assume the Tensorflow format, where the last entry denotes the number of channels.

Returns
The output of the network.

Return type
y

training: bool

CHAPTER

FIVE

UTILITIES AND HELPER FUNCTIONS

Contents

This subpackage contains common helper functions to a variety of problems (e.g., PyTorch checkpointing, special layers, computing diagonal Fisher matrices, ...).

5.1 Batch Normalization

Implementation of a hypernet compatible batchnorm layer.

The joint use of batch-normalization and hypernetworks is not straight forward, mainly due to the statistics accumulated by the batch-norm operation which expect the weights of the main network to only change slowly. If a hypernetwork replaces the whole set of weights, the statistics previously estimated by the batch-norm layer might be completely off.

To circumvent this problem, we provide multiple solutions:

- In a continual learning setting with one set of weights per task, we can simply estimate and store statistics per task (hence, the batch-norm operation has to be conditioned on the task).
- The statistics are distilled into the hypernetwork. This would require the addition of an extra loss term.
- The statistics can be treated as parameters that are outputted by the hypernetwork. In this case, nothing enforces that these "statistics" behave similar to statistics that would result from a running estimate (hence, the resulting operation might have nothing in common with batch- norm).
- Always use the statistics estimated on the current batch.

Note, we also provide the option of turning off the statistics, in which case the statistics will be set to zero mean and unit variance. This is helpful when interpreting batch-normalization as a general form of gain modulation (i.e., just applying a shift and scale to neural activities).

class hypnettorch.utils.batchnorm_layer.**BatchNormLayer**(*num_features*, *momentum=0.1*, *affine=True*, *track_running_stats=True*, *frozen_stats=False*, *learnable_stats=False*)

Bases: Module

Hypernetwork-compatible batch-normalization layer.

Note, batch normalization performs the following operation

$$y = \frac{x - \mathrm{E}[x]}{\sqrt{\mathrm{Var}[x] + \epsilon}} * \gamma + \beta$$

This class allows to deviate from this standard implementation in order to provide the flexibility required when using hypernetworks. Therefore, we slightly change the notation to

$$y = \frac{x - m_{\text{stats}}^{(t)}}{\sqrt{v_{\text{stats}}^{(t)} + \epsilon}} * \gamma^{(t)} + \beta^{(t)}$$

We use this notation to highlight that the running statistics $m_{\text{stats}}^{(t)}$ and $v_{\text{stats}}^{(t)}$ are not necessarily estimates resulting from mean and variance computation but might be learned parameters (e.g., the outputs of a hypernetwork).

We additionally use the superscript (t) to denote that the gain γ, offset β and statistics may be dynamically selected based on some external context information.

This class provides the possibility to checkpoint statistics $m_{\text{stats}}^{(t)}$ and $v_{\text{stats}}^{(t)}$, but **not** gains and offsets.

Note: If context-dependent gains $\gamma^{(t)}$ and offsets $\beta^{(t)}$ are required, then they have to be maintained externally, e.g., via a task-conditioned hypernetwork (see this paper for an example) and passed to the *forward()* method.

Parameters

- **num_features** – See argument num_features, for instance, of class torch.nn.BatchNorm1d.

- **momentum** – See argument momentum of class torch.nn.BatchNorm1d.
- **affine** – See argument affine of class torch.nn.BatchNorm1d. If set to False, the input activity will simply be "whitened" according to the applied layer statistics (except if gain γ and offset β are passed to the *forward()* method).

  Note, if learnable_stats is False, then setting affine to False results in no learnable weights for this layer (running stats might still be updated, but not via gradient descent).

  Note, even if this option is False, one may still pass a gain γ and offset β to the *forward()* method.
- **track_running_stats** – See argument track_running_stats of class torch.nn.BatchNorm1d.
- **frozen_stats** – If True, the layer statistics are frozen at their initial values of $\gamma = 1$ and $\beta = 0$, i.e., layer activity will not be whitened.

  Note, this option requires track_running_stats to be set to False.
- **learnable_stats** – If True, the layer statistics are initialized as learnable parameters (requires_grad=True).

  Note, these extra parameters will be maintained internally and not added to the *weights*. Statistics can always be maintained externally and passed to the *forward()* method.

  Note, this option requires track_running_stats to be set to False.

checkpoint_stats(*device=None*)

Buffers for a new set of running stats will be registered.

Calling this function will also increment the attribute *num_stats*.

Parameters

device (*optional*) – If not provided, the newly created statistics will either be moved to the device of the most recent statistics or to CPU if no prior statistics exist.

forward(*inputs*, *running_mean=None*, *running_var=None*, *weight=None*, *bias=None*, *stats_id=None*)

Apply batch normalization to given layer activations.

Based on the state if this module (attribute *training*), the configuration of this layer and the parameters currently passed, the behavior of this function will be different.

The core of this method still relies on the function torch.nn.functional.batch_norm(). In the following we list the different behaviors of this method based on the context.

In training mode:

We first consider the case that this module is in training mode, i.e., torch.nn.Module.train() has been called.

Usually, during training, the running statistics are not used when computing the output, instead the statistics computed on the current batch are used (denoted by *use batch stats* in the table below). However, the batch statistics are typically updated during training (denoted by *update running stats* in the table below).

The above described scenario would correspond to passing batch statistics to the function torch.nn.functional.batch_norm() and setting the parameter training to True.

training mode	use batch stats	update running stats
given stats	Yes	Yes
track running stats	Yes	Yes
frozen stats	No	No
learnable stats	Yes	Yes[1]
no track running stats	Yes	No

The meaning of each row in this table is as follows:

- **given stats**: External stats are provided via the parameters `running_mean` and `running_var`.
- **track running stats**: If `track_running_stats` was set to `True` in the constructor and no stats were given.
- **frozen stats**: If `frozen_stats` was set to `True` in the constructor and no stats were given.
- **learnable stats**: If `learnable_stats` was set to `True` in the constructor and no stats were given.
- **no track running stats**: If none of the above options apply, then the statistics will always be computed from the current batch (also in eval mode).

Note: If provided, running stats specified via `running_mean` and `running_var` always have priority.

In evaluation mode:

We now consider the case that this module is in evaluation mode, i.e., `torch.nn.Module.eval()` has been called.

Here is the same table as above just for the evaluation mode.

evaluation mode	use batch stats	update running stats
track running stats	No	No
frozen stats	No	No
learnable stats	No	No
given stats	No	No
no track running stats	Yes	No

Parameters

- **inputs** – The inputs to the batchnorm layer.
- **running_mean** (*optional*) – Running mean stats m_{stats}. This option has priority, i.e., any internally maintained statistics are ignored if given.

  Note: If specified, then `running_var` also has to be specified.

- **running_var** (*optional*) – Similar to option `running_mean`, but for the running variance stats v_{stats}

  Note: If specified, then `running_mean` also has to be specified.

[1] We use a custom implementation to update the running statistics, that is compatible with backpropagation.

- **weight** (*optional*) – The gain factors γ. If given, any internal gains are ignored. If option `affine` was set to `False` in the constructor and this option remains `None`, then no gains are multiplied to the "whitened" inputs.
- **bias** (*optional*) – The behavior of this option is similar to option `weight`, except that this option represents the offsets β.
- **stats_id** – This argument is optional except if multiple running stats checkpoints exist (i.e., attribute *num_stats* is greater than 1) and no running stats have been provided to this method.

  Note: This argument is ignored if running stats have been passed.

Returns
The layer activation `inputs` after batch-norm has been applied.

get_stats(*stats_id=None*)

Get a set of running statistics (means and variances).

Parameters
stats_id (*optional*) – ID of stats. If not provided, the most recent stats are returned.

Returns

Tuple containing:

- **running_mean**
- **running_var**

Return type
(tuple)

property hyper_shapes

A list of list of integers. Each list represents the shape of a weight tensor that can be passed to the *forward()* method. If all weights are maintained internally, then this attribute will be `None`.

Specifically, this attribute is controlled by the argument `affine`. If `affine` is `True`, this attribute will be `None`. Otherwise this attribute contains the shape of γ and β.

Type
list or None

property num_stats

The number T of internally managed statistics $\{(m_{\text{stats}}^{(1)}, v_{\text{stats}}^{(1)}), \ldots, (m_{\text{stats}}^{(T)}, v_{\text{stats}}^{(T)})\}$. This number is incremented everytime the method *checkpoint_stats()* is called.

Type
int

property param_shapes

A list of list of integers. Each list represents the shape of a parameter tensor.

Note, this attribute is independent of the attribute *weights*, it always comprises the shapes of all weight tensors as if the network would be stand-alone (i.e., no weights being passed to the *forward()* method). Note, unless `learnable_stats` is enabled, the layer statistics are not considered here.

Type
list

training: bool

property weights

A list of all internal weights of this layer. If all weights are assumed to be generated externally, then this attribute will be None.

Type
list or None

5.2 Common command-line arguments

This file has a collection of helper functions that can be used to specify command-line arguments. In particular, arguments that are necessary for multiple experiments (even though with different default values) should be specified here, such that we do not define arguments (and their help texts) multiple times.

All functions specified here are helper functions for a simulation specific argument parser such as `cifar.train_args.parse_cmd_arguments()`.

5.2.1 Important note for contributors

DO NEVER CHANGE DEFAULT VALUES. Instead, add a keyword argument to the corresponding method, that allows you to change the default value, when you call the method.

hypnettorch.utils.cli_args.**check_invalid_argument_usage**(*args*)

This method checks for common conflicts when using the arguments defined by methods in this module.

The following things will be checked:

- Based on the optimizer choices specified in *train_args()*, we assert here that only one optimizer is selected at a time.
- Assert that *clip_grad_value* and *clip_grad_norm* are not set at the same time.
- Assert that *split_head_cl3* is only set for *cl_scenario=3*
- Assert that the arguments specified in function *main_net_args()* are correctly used.

Note: The checks can't handle prefixes yet.

Parameters
args – The parsed command-line arguments, i.e., the output of method argparse.ArgumentParser.parse_args().

Raises
ValueError – If invalid argument combinations are used.

hypnettorch.utils.cli_args.**cl_args**(*parser*, *show_beta=True*, *dbeta=0.01*, *show_from_scratch=False*, *show_multi_head=False*, *show_cl_scenario=False*, *show_split_head_cl3=True*, *dcl_scenario=1*, *show_num_tasks=False*, *dnum_tasks=1*, *show_num_classes_per_task=False*, *dnum_classes_per_task=2*, *show_calc_hnet_reg_targets_online=False*, *show_hnet_reg_batch_size=False*, *dhnet_reg_batch_size=-1*)

This is a helper method of the method *parse_cmd_arguments* to add an argument group for typical continual learning arguments.

Arguments specified in this function:

- *beta*
- *train_from_scratch*
- *multi_head*
- *cl_scenario*
- *split_head_cl3*
- *num_tasks*
- *num_classes_per_task*
- *calc_hnet_reg_targets_online*
- *hnet_reg_batch_size*

Parameters

- **parser** – Object of class `argparse.ArgumentParser`.
- **show_beta** – Whether option *beta* should be shown.
- **dbeta** – Default value of option *beta*.
- **show_from_scratch** – Whether option *train_from_scratch* should be shown.
- **show_multi_head** – Whether option *multi_head* should be shown.
- **show_cl_scenario** – Whether option *cl_scenario* should be shown.
- **show_split_head_cl3** – Whether option *split_head_cl3* should be shown. Only has an effect if `show_cl_scenario` is `True`.
- **dcl_scenario** – Default value of option *cl_scenario*.
- **show_num_tasks** – Whether option *num_tasks* should be shown.
- **dnum_tasks** – Default value of option *num_tasks*.
- **show_num_classes_per_task** – Whether option *show_num_classes_per_task* should be shown.
- **dnum_classes_per_task** – Default value of option *dnum_classes_per_task*.
- **show_calc_hnet_reg_targets_online** (*bool*) – Whether the option *calc_hnet_reg_targets_online* should be provided.
- **show_hnet_reg_batch_size** (*bool*) – Whether the option *hnet_reg_batch_size* should be provided.
- **dhnet_reg_batch_size** (*int*) – Default value of option *hnet_reg_batch_size*.

Returns

The created argument group, in case more options should be added.

hypnettorch.utils.cli_args.**data_args**(*parser*, *show_disable_data_augmentation=False*, *show_data_dir=False*, *ddata_dir='.'*)

This is a helper method of the function *parse_cmd_arguments* to add an argument group for typical dataset related options.

Arguments specified in this function:

- *disable_data_augment*

Parameters

- **parser** – Object of class argparse.ArgumentParser.
- **show_disable_data_augmentation** (*bool*) – Whether option *disable_data_augmentation* should be shown.
- **show_data_dir** (*bool*) – Whether option *data_dir* should be shown.
- **ddata_dir** (*str*) – Default value of option *data_dir*.

Returns

The created argument group, in case more options should be added.

hypnettorch.utils.cli_args.**eval_args**(*parser*, *dval_iter=500*, *show_val_batch_size=False*, *dval_batch_size=256*, *show_val_set_size=False*, *dval_set_size=0*, *show_test_with_val_set=False*)

This is a helper method of the method *parse_cmd_arguments* to add an argument group for validation and testing options.

Arguments specified in this function:

- *val_iter*
- *val_batch_size*
- *val_set_size*
- *test_with_val_set*

Parameters

- **parser** – Object of class argparse.ArgumentParser.
- **dval_iter** (*int*) – Default value of argument *val_iter*.
- **show_val_batch_size** (*bool*) – Whether the *val_batch_size* argument should be shown.
- **dval_batch_size** (*int*) – Default value of argument *val_batch_size*.
- **show_val_set_size** (*bool*) – Whether the *val_set_size* argument should be shown.
- **dval_set_size** (*int*) – Default value of argument *val_set_size*.
- **show_test_with_val_set** (*bool*) – Whether the *test_with_val_set* argument should be shown.

Returns

The created argument group, in case more options should be added.

hypnettorch.utils.cli_args.**gan_args**(*parser*)

This is a helper method of the method *parse_cmd_arguments* to add an argument group for options to configure the generator and discriminator network.

Deprecated since version 1.0: Please use method *main_net_args()* and *generator_args()* instead.

Parameters

parser – Object of class argparse.ArgumentParser.

Returns
The created argument group, in case more options should be added.

hypnettorch.utils.cli_args.**generator_args**(*agroup*, *dlatent_dim=3*)

This is a helper method of the method *parse_cmd_arguments* (or more specifically an auxillary method to *train_args()*) to add arguments to an argument group for options specific to a main network that should act as a generator.

Arguments specified in this function:

- *latent_dim*
- *latent_std*

Parameters

- **agroup** – The argument group returned by, for instance, function *main_net_args()*.
- **dlatent_dim** – Default value of option *latent_dim*.

hypnettorch.utils.cli_args.**hnet_args**(*parser*, *allowed_nets=['hmlp']*, *dhmlp_arch='100,100'*, *show_cond_emb_size=True*, *dcond_emb_size='8'*, *dchmlp_chunk_size=1000*, *dchunk_emb_size=8*, *show_use_cond_chunk_embs=True*, *dhdeconv_shape='512,512,3'*, *prefix=None*, *pf_name=None*, ***kwargs*)

This is a helper function to add an argument group for hypernetwork- specific arguments to a given argument parser.

Arguments specified in this function:

- *hnet_type*
- *hmlp_arch*
- *cond_emb_size*
- *chmlp_chunk_size*
- *chunk_emb_size*
- *use_cond_chunk_embs*
- *hdeconv_shape*
- *hdeconv_num_layers*
- *hdeconv_filters*
- *hdeconv_kernels*
- *hdeconv_attention_layers*

Parameters

- **parser** (*argparse.ArgumentParser*) – The parser to which an argument group should be added
- **allowed_nets** (*list*) – List of allowed network identifiers. The following identifiers are considered (note, we also reference the network that each network type targets):
  - `'hmlp'`: `hnets.mlp_hnet.HMLP`
  - `'chunked_hmlp'`: `hnets.chunked_mlp_hnet.ChunkedHMLP`
  - `'structured_hmlp'`: `hnets.structured_mlp_hnet.StructuredHMLP`

  - 'hdeconv': hnets.deconv_hnet.HDeconv
  - 'chunked_hdeconv': hnets.chunked_deconv_hnet.ChunkedHDeconv
- **dhmlp_arch** (*str*) – Default value of option *hmlp_arch*.
- **show_cond_emb_size** (*bool*) – Whether the option *cond_emb_size* should be provided.
- **dcond_emb_size** (*int*) – Default value of option *cond_emb_size*.
- **dchmlp_chunk_size** (*int*) – Default value of option *chmlp_chunk_size*.
- **dchunk_emb_size** (*int*) – Default value of option *chunk_emb_size*.
- **show_use_cond_chunk_embs** (*bool*) – Whether the option *use_cond_chunk_embs* should be provided (if applicable to network types).
- **dhdeconv_shape** (*str*) – Default value of option *hdeconv_shape*.
- **prefix** (*str, optional*) – If arguments should be instantiated with a certain prefix. E.g., a setup requires several hypernetworks, that may need different settings. For instance: `prefix='gen_'`.
- **pf_name** (*str, optional*) – A name of type of hypernetwork for which that `prefix` is needed. For instance: `prefix='generator'`.
- ****kwargs** – Keyword arguments to configure options that are common across main networks (note, a hypernet is just a special main network). See arguments of *main_net_args()*.

Returns
The created argument group containing the desired options.

Return type
(argparse._ArgumentGroup)

hypnettorch.utils.cli_args.**init_args**(*parser*, *custom_option=True*, *show_normal_init=True*, *show_hyper_fan_init=False*)

This is a helper method of the method *parse_cmd_arguments* to add an argument group for options regarding network initialization.

Arguments specified in this function:

- *custom_network_init*
- *normal_init*
- *std_normal_init*
- *std_normal_temb*
- *std_normal_emb*
- *hyper_fan_init*

Parameters

- **parser** – Object of class argparse.ArgumentParser.
- **custom_option** (*bool*) – Whether the option *custom_network_init* should be provided.
- **show_normal_init** (*bool*) – Whether the option *normal_init* and *std_normal_init* should be provided.
- **show_hyper_fan_init** (*bool*) – Whether the option *hyper_fan_init* should be provided.

Returns

The created argument group, in case more options should be added.

hypnettorch.utils.cli_args.**main_net_args**(*parser*, *allowed_nets=['mlp']*, *dmlp_arch='100,100'*, *dlenet_type='mnist_small'*, *dcmlp_arch='10,10'*, *dcmlp_chunk_arch='10,10'*, *dcmlp_in_cdim=100*, *dcmlp_out_cdim=10*, *dcmlp_cemb_dim=8*, *dresnet_block_depth=5*, *dresnet_channel_sizes='16,16,32,64'*, *dwrn_block_depth=4*, *dwrn_widening_factor=10*, *diresnet_channel_sizes='64,64,128,256,512'*, *diresnet_blocks_per_group='2,2,2,2'*, *dsrnn_rec_layers='10'*, *dsrnn_pre_fc_layers=''*, *dsrnn_post_fc_layers=''*, *dsrnn_rec_type='lstm'*, *show_net_act=True*, *dnet_act='relu'*, *show_no_bias=False*, *show_dropout_rate=True*, *ddropout_rate=-1*, *show_specnorm=True*, *show_batchnorm=True*, *show_no_batchnorm=False*, *show_bn_no_running_stats=False*, *show_bn_distill_stats=False*, *show_bn_no_stats_checkpointing=False*, *prefix=None*, *pf_name=None*)

This is a helper function for the function *parse_cmd_arguments* to add an argument group for options to a main network.

Arguments specified in this function:

- *net_type*
- *fc_arch*
- *mlp_arch*
- *lenet_type*
- *cmlp_arch*
- *cmlp_chunk_arch*
- *cmlp_in_cdim*
- *cmlp_out_cdim*
- *cmlp_cemb_dim*
- *resnet_block_depth*
- *resnet_channel_sizes*
- *wrn_block_depth*
- *wrn_widening_factor*
- *wrn_use_fc_bias*
- *iresnet_use_fc_bias*
- *iresnet_channel_sizes*
- *iresnet_blocks_per_group*
- *iresnet_bottleneck_blocks*
- *iresnet_projection_shortcut*
- *srnn_rec_layers*

- *srnn_pre_fc_layers*
- *srnn_post_fc_layers*
- *srnn_no_fc_out*
- *srnn_rec_type*
- *net_act*
- *no_bias*
- *dropout_rate*
- *specnorm*
- *batchnorm*
- *no_batchnorm*
- *bn_no_running_stats*
- *bn_distill_stats*
- *bn_no_stats_checkpointing*

Parameters

- **parser** (argparse.ArgumentParser) – The argument parser to which the argument group should be added.
- **allowed_nets** (*list*) – List of allowed network identifiers. The following identifiers are considered (note, we also reference the network that each network type targets):
  - mlp: mnets.mlp.MLP
  - lenet: mnets.lenet.LeNet
  - resnet: mnets.resnet.ResNet
  - wrn: mnets.wide_resnet.WRN
  - iresnet: mnets.resnet_imgnet.ResNetIN
  - zenke: mnets.zenkenet.ZenkeNet
  - bio_conv_net: mnets.bio_conv_net.BioConvNet
  - chunked_mlp: mnets.chunk_squeezer.ChunkSqueezer
  - simple_rnn: mnets.simple_rnn.SimpleRNN
- **dmlp_arch** – Default value of option *mlp_arch*.
- **dlenet_type** – Default value of option *lenet_type*.
- **dcmlp_arch** – Default value of option *cmlp_arch*.
- **dcmlp_chunk_arch** – Default value of option *cmlp_chunk_arch*.
- **dcmlp_in_cdim** – Default value of option *cmlp_in_cdim*.
- **dcmlp_out_cdim** – Default value of option *cmlp_out_cdim*.
- **dcmlp_cemb_dim** – Default value of option *cmlp_cemb_dim*.
- **dresnet_block_depth** – Default value of option *resnet_block_depth*.
- **dresnet_channel_sizes** – Default value of option *resnet_channel_sizes*.

- **dwrn_block_depth** – Default value of option *wrn_block_depth*.
- **dwrn_widening_factor** – Default value of option *wrn_widening_factor*.
- **diresnet_channel_sizes** – Default value of option *iresnet_channel_sizes*.
- **diresnet_blocks_per_group** – Default value of option *iresnet_blocks_per_group*.
- **dsrnn_rec_layers** – Default value of option *srnn_rec_layers*.
- **dsrnn_pre_fc_layers** – Default value of option *srnn_pre_fc_layers*.
- **dsrnn_post_fc_layers** – Default value of option *srnn_post_fc_layers*.
- **dsrnn_rec_type** – Default value of option *srnn_rec_type*.
- **show_net_act** (*bool*) – Whether the option *net_act* should be provided.
- **dnet_act** – Default value of option *net_act*.
- **show_no_bias** (*bool*) – Whether the option *no_bias* should be provided.
- **show_dropout_rate** (*bool*) – Whether the option *dropout_rate* should be provided.
- **ddropout_rate** – Default value of option `dropout_rate`.
- **show_specnorm** (*bool*) – Whether the option *specnorm* should be provided.
- **show_batchnorm** (*bool*) – Whether the option *batchnorm* should be provided.
- **show_no_batchnorm** (*bool*) – Whether the option *no_batchnorm* should be provided.
- **show_bn_no_running_stats** (*bool*) – Whether the option *bn_no_running_stats* should be provided.
- **show_bn_distill_stats** (*bool*) – Whether the option *bn_distill_stats* should be provided.
- **show_bn_no_stats_checkpointing** (*bool*) – Whether the option *bn_no_stats_checkpointing* should be provided.
- **prefix** (*optional*) – If arguments should be instantiated with a certain prefix. E.g., a setup requires several main network, that may need different settings. For instance: prefix=:code:*prefix='gen_'*.
- **pf_name** (*optional*) – A name of the type of main net for which that prefix is needed. For instance: prefix=:code:*'generator'*.

Returns

The created argument group, in case more options should be added.

hypnettorch.utils.cli_args.**miscellaneous_args**(*parser*, *big_data=True*, *synthetic_data=False*, *show_plots=False*, *no_cuda=False*, *dout_dir=None*, *show_publication_style=False*)

This is a helper method of the method *parse_cmd_arguments* to add an argument group for miscellaneous arguments.

Arguments specified in this function:

- *num_workers*
- *out_dir*
- *use_cuda*
- *no_cuda*

- *loglevel_info*
- *deterministic_run*
- *publication_style*
- *show_plots*
- *data_random_seed*
- *random_seed*

Parameters

- **parser** – Object of class argparse.ArgumentParser.
- **big_data** – If the program processes big datasets that need to be loaded from disk on the fly. In this case, more options are provided.
- **synthetic_data** – If data is randomly generated, then we want to decouple this randomness from the training randomness.
- **show_plots** – Whether the option *show_plots* should be provided.
- **no_cuda** – If True, the user has to explicitly set the flag *–use_cuda* rather than using CUDA by default.
- **dout_dir** (*optional*) – Default value of option *out_dir*. If None, the default value will be *./out/run_<YY>-<MM>-<DD>_<hh>-<mm>-<ss>* that contains the current date and time.
- **show_publication_style** – Whether the option *publication_style* should be provided.

Returns

The created argument group, in case more options should be added.

hypnettorch.utils.cli_args.**train_args**(*parser*, *show_lr=False*, *dlr=0.1*, *show_epochs=False*, *depochs=-1*, *dbatch_size=32*, *dn_iter=100001*, *show_use_adam=False*, *dadam_beta1=0.9*, *show_use_rmsprop=False*, *show_use_adadelta=False*, *show_use_adagrad=False*, *show_clip_grad_value=False*, *show_clip_grad_norm=False*, *show_adam_beta1=False*, *show_momentum=True*)

This is a helper method of the method *parse_cmd_arguments* to add an argument group for options to configure network training.

Arguments specified in this function:

- *batch_size*
- *n_iter*
- *epochs*
- *lr*
- *momentum*
- *weight_decay*
- *use_adam*
- *adam_beta1*
- *use_rmsprop*
- *use_adadelta*

- *use_adagrad*
- *clip_grad_value*
- *clip_grad_norm*

Parameters

- **parser** – Object of class `argparse.ArgumentParser`.
- **show_lr** – Whether the *lr* - learning rate - argument should be shown. Might not be desired if individual learning rates per optimizer should be specified.
- **dlr** – Default value for option *lr*.
- **show_epochs** – Whether the *epochs* argument should be shown.
- **depochs** – Default value for option *epochs*.
- **dbatch_size** – Default value for option *batch_size*.
- **dn_iter** – Default value for option *n_iter*.
- **show_use_adam** – Whether the *use_adam* argument should be shown. Will also show the *adam_beta1* argument.
- **dadam_beta1** – Default value for option *adam_beta1*.
- **show_use_rmsprop** – Whether the *use_rmsprop* argument should be shown.
- **show_use_adadelta** – Whether the *use_adadelta* argument should be shown.
- **show_use_adagrad** – Whether the *use_adagrad* argument should be shown.
- **show_clip_grad_value** – Whether the *clip_grad_value* argument should be shown.
- **show_clip_grad_norm** – Whether the *clip_grad_norm* argument should be shown.
- **show_adam_beta1** – Whether the *adam_beta1* argument should be shown. Note, this argument is also shown when `show_use_adam` is `True`.
- **show_momentum** – Whether the *momentum* argument should be shown.

Returns

The created argument group, in case more options should be added.

5.3 Context-modulation layer

This module should represent a special gain-modulation layer that can modulate neural computation based on an external context.

class hypnettorch.utils.context_mod_layer.**ContextModLayer**(*num_features*, *no_weights=False*, *no_gains=False*, *no_shifts=False*, *apply_gain_offset=False*, *apply_gain_softplus=False*, *softplus_scale=1.0*)

Bases: `Module`

Implementation of a layer that can apply context-dependent modulation on the level of neuronal computation.

The layer consists of two parameter vectors: gains $\mathbf{g}$ and shifts $\mathbf{s}$, whereas gains represent a multiplicative modulation of input activations and shifts an additive modulation, respectively.

Note, the weight vectors $\mathbf{g}$ and $\mathbf{s}$ might also be passed to the *forward()* method, where one may pass a separate set of parameters for each sample in the input batch.

Example

Assume that a *ContextModLayer* is applied between a linear (fully-connected) layer $\mathbf{y} \equiv W\mathbf{x} + \mathbf{b}$ with input $\mathbf{x}$ and a nonlinear activation function $z \equiv \sigma(y)$.

The layer-computation in such a case will become

$$\sigma\big((W\mathbf{x} + \mathbf{b}) \odot \mathbf{g} + \mathbf{s}\big)$$

Parameters

- **num_features** (*int or tuple*) – Number of units in the layer (size of parameter vectors $\mathbf{g}$ and $\mathbf{s}$).

  In case a `tuple` of integers is provided, the gain $\mathbf{g}$ and shift $\mathbf{s}$ parameters will become multidimensional tensors with the shape being prescribed by `num_features`. Please note the broadcasting rules as $\mathbf{g}$ and $\mathbf{s}$ are simply multiplied or added to the input.

  #### Example

  Consider the output of a convolutional layer with output shape `[B,C,W,H]`. In case there should be a scalar gain and shift per feature map, `num_features` could be `[C,1,1]` or `[1, C,1,1]` (one might also pass a shape `[B,C,1,1]` to the *forward()* method to apply separate shifts and gains per sample in the batch).

  Alternatively, one might want to provide shift and gain per output unit, i.e., `num_features` should be `[C,W,H]`. Note, that due to weight sharing, all output activities within a feature map are computed using the same weights, which is why it is common practice to share shifts and gains within a feature map (e.g., in Spatial Batch-Normalization).

- **no_weights** (*bool*) – If `True`, the layer will have no trainable weights ($\mathbf{g}$ and $\mathbf{s}$). Hence, weights are expected to be passed to the *forward()* method.

- **no_gains** (*bool*) – If `True`, no gain parameters $\mathbf{g}$ will be modulating the input activity.

  Note: Arguments `no_gains` and `no_shifts` might not be activated simultaneously!

- **no_shifts** (*bool*) – If `True`, no shift parameters $\mathbf{s}$ will be modulating the input activity.

- **apply_gain_offset** (*bool, optional*) – If activated, this option will apply a constant offset of 1 to all gains, i.e., the computation becomes

  $$\sigma\big((W\mathbf{x} + \mathbf{b}) \odot (1 + \mathbf{g}) + \mathbf{s}\big)$$

  When could that be useful? In case the gains and shifts are generated by the same hypernetwork, a meaningful initialization might be difficult to achieve (e.g., such that gains are close to 1 and shifts are close to 0 at the beginning). Therefore, one might initialize the hypernetwork such that all outputs are close to zero at the beginning and the constant shift ensures that meaningful gains are applied.

- **apply_gain_softplus** (*bool, optional*) – If activated, this option will enforce poitive gain modulation by sending the gain weights $\mathbf{g}$ through a softplus function (scaled by s, see `softplus_scale`).

$$\mathbf{g} = \frac{1}{s} \log(1 + \exp(\mathbf{g} \cdot s))$$

- **softplus_scale** (*float*) – If option `apply_gain_softplus` is `True`, then this will determine the sclae of the softplus function.

Initializes internal Module state, shared by both nn.Module and ScriptModule.

checkpoint_weights(*device=None*, *no_reinit=False*)

Checkpoint and reinit the current weights.

Buffers for a new checkpoint will be registered and the current weights will be copied into them. Additionally, the current weights will be reinitialized (gains to 1 and shifts to 0).

Calling this function will also increment the attribute *num_ckpts*.

Note: This method uses the method `torch.nn.Module.register_buffer()` rather than the method `torch.nn.Module.register_parameter()` to create checkpoints. The reason is, that we don't want the checkpoints to appear as trainable weights (when calling `torch.nn.Module.parameters()`). However, that means that training on checkpointed weights cannot be continued unless they are copied back into an actual `torch.nn.Parameter` object.

Parameters

- **device** (*optional*) – If not provided, the newly created checkpoint will be moved to the device of the current weights.
- **no_reinit** (*bool*) – If `True`, the actual *weights* will not be reinitialized.

forward(*x*, *weights=None*, *ckpt_id=None*, *bs_dim=0*)

Apply context-dependent gain modulation.

Computes $\mathbf{x} \odot \mathbf{g} + \mathbf{s}$, where $\mathbf{x}$ denotes the input activity `x`.

Parameters

- **x** – The input activity.
- **weights** – Weights that should be used instead of the internally maintained once (determined by attribute *weights*). Note, if `no_weights` was `True` in the constructor, then this parameter is mandatory.

  Usually, the shape of the passed weights should follow the attribute *param_shapes*, which is a tuple of shapes `[[num_features], [num_features]]` (at least for linear layers, see docstring of argument `num_features` in the constructor for more details). However, one may also specify a seperate set of context-mod parameters per input sample. Assume `x` has shape `[num_samples, num_features]`. Then `weights` may have the shape `[[num_samples, num_features], [num_samples, num_features]]`.
- **ckpt_id** (*int*) – This argument can be set in case a checkpointed set of weights should be used to compute the forward pass (see method *checkpoint_weights()*).

  Note: This argument is ignored if `weights` is not `None`.

- **bs_dim** (*int*) – Batch size dimension in input tensor `x`.

Returns
The modulated input activity.

property gain_offset_applied

Whether constructor argument `apply_gain_offset` was activated.

Thus, whether an offset for the gain `g` is applied.

Type
bool

property gain_softplus_applied

Whether constructor argument `apply_gain_softplus` was activated.

Thus, whether a softplus function for the gain `g` is applied.

Type
bool

get_weights(*ckpt_id=None*)

Get the current (or a set of checkpointed) weights of this context- mod layer.

Parameters
ckpt_id (*optional*) – ID of checkpoint. If not provided, the current set of weights is returned. If `ckpt_id == self.num_ckpts`, then this method also returns the current weights, as the checkpoint has not been created yet.

Returns

Tuple containing:

- **gain**: Is `None` if layer has no gains.
- **shift**: Is `None` if layer has no shifts.

Return type
(tuple)

property has_gains

Is `True` if `no_gains` was not set in the constructor.

Thus, whether gains `g` are part of the computation of this layer.

Type
bool

property has_shifts

Is `True` if `no_shifts` was not set in the constructor.

Thus, whether shifts `s` are part of the computation of this layer.

Type
bool

normal_init(*std=1.0*)

Reinitialize internal weights using a normal distribution.

Parameters
std (*float*) – Standard deviation of init.

property num_ckpts

The number of existing weight checkpoints (i.e., how often the method *checkpoint_weights()* was called).

Type
int

property param_shapes

A list of list of integers. Each list represents the shape of a parameter tensor. Note, this attribute is independent of the attribute *weights*, it always comprises the shapes of all weight tensors as if the network would be stand- alone (i.e., no weights being passed to the *forward()* method).

Note: The weights passed to the *forward()* method might deviate from these shapes, as we allow passing a distinct set of parameters per sample in the input batch.

Type
list

property param_shapes_meta

List of strings. Each entry represents the meaning of the corresponding entry in *param_shapes*. The following keywords are possible:

- `'gain'`: The corresponding shape in *param_shapes* denotes the gain `g` parameter.
- `'shift'`: The corresponding shape in *param_shapes* denotes the shift `s` parameter.

Type
list

preprocess_gain(*gain*)

Obtains gains `g` used for mudulation.

Depending on the user configuration, gains might be preprocessed before applied for context-modulation (e.g., see attributes *gain_offset_applied* or *gain_softplus_applied*). This method transforms raw gains such that they can be applied to the network activation.

Note: This method is called by the *forward()* to transform given gains.

Parameters
gain (*torch.Tensor*) – A gain tensor.

Returns
The transformed gains.

Return type
(torch.Tensor)

sparse_init(*sparsity=0.8*)

Reinitialize internal weights sparsely.

Gains will be initialized such that `sparisity * 100` percent of them will be 0, the remaining ones will be 1. Shifts are initialized to 0.

Parameters
sparsity (*float*) – A number between 0 and 1 determining the spasity level of gains.

training: bool

uniform_init(*width=1.0*)

Reinitialize internal weights using a uniform distribution.

Parameters
width (*float*) – The range of the uniform init will be determined as [mean-width, mean+width], where mean is 0 for shifts and 1 for gains.

property weights

A list of all internal weights of this layer.

If all weights are assumed to be generated externally, then this attribute will be None.

Type
torch.nn.ParameterList or None

5.4 Elastic Weight Consolidation

Implementation of EWC:
https://arxiv.org/abs/1612.00796

Note, these implementation are based on the descriptions provided in:
https://arxiv.org/abs/1809.10635

The code is inspired by the corresponding implementation:
https://git.io/fjcnL

hypnettorch.utils.ewc_regularizer.**compute_fisher**(*task_id*, *data*, *params*, *device*, *mnet*, *hnet=None*, *empirical_fisher=True*, *online=False*, *gamma=1.0*, *n_max=-1*, *regression=False*, *time_series=False*, *allowed_outputs=None*, *custom_forward=None*, *custom_nll=None*, *pass_ids=False*, *proper_scaling=False*, *prior_strength=None*, *regression_lvar=1.0*, *target_manipulator=None*)

Compute estimates of the diagonal elements of the Fisher information matrix, as needed as importance-weights by elastic weight consolidation (EWC).

The Fisher matrix for a conditional distribution $p(y \mid \theta, x)$ (i.e., the model likelihood for a model with parameters θ) is defined as follows at location x

$$\begin{aligned}\mathcal{F}(x) &= \text{Var}\big[\nabla_\theta \log p(y \mid \theta, x)\big] \\ &= \mathbb{E}_{p(y|\theta,x)}\big[\nabla_\theta \log p(y \mid \theta, x) \nabla_\theta \log p(y \mid \theta, x)^T\big]\end{aligned}$$

In practice, we are often interested in the Fisher averaged over locations

$$\mathcal{F} = \mathbb{E}_{p(x)}[\mathcal{F}(x)]$$

Since the model is trained, such that in-distribution the model likelihood $p(y \mid \theta, x)$ and the ground-truth likelihood $p(y \mid x)$ agree, people often refer to the empirical Fisher, which utilizes the dataset for computation and therewith doesn't require sampling from the model likelihood. Note, EWC anyway assumes that in-distribution

$p(y \mid \theta, x) = p(y \mid x)$ in order to be able to replace the Hessian by the Fisher matrix.

$$\begin{aligned}\mathcal{F}_{emp} &= \mathbb{E}_{p(x,y)}\big[\nabla_\theta \log p(y \mid \theta, x) \nabla_\theta \log p(y \mid \theta, x)^T\big] \\ &= \mathbb{E}_{p(x)}\Big[\mathbb{E}_{p(y|x)}\big[\nabla_\theta \log p(y \mid \theta, x) \nabla_\theta \log p(y \mid \theta, x)^T\big]\Big] \\ &\approx \frac{1}{|\mathcal{D}|} \sum_{(x_n, y_n) \sim \mathcal{D}} \big[\nabla_\theta \log p(y_n \mid \theta, x_n) \nabla_\theta \log p(y_n \mid \theta, x_n)^T\big]\Big]\end{aligned}$$

Note: This method registers buffers in the given module (storing the current parameters and the estimate of the Fisher diagonal elements), i.e., the `mnet` if `hnet` is `None`, otherwise the `hnet`.

Parameters

- **task_id** – The ID of the current task, needed to store the computed tensors with a unique name. When `hnet` is given, it is used as input to the `hnet` forward method to select the current task embedding.
- **data** – A data handler. We will compute the Fisher estimate across the whole training set (except `n_max` is specified).
- **params** – A list of parameter tensors from the module of which we aim to compute the Fisher for. If `hnet` is given, then these are assumed to be the "theta" parameters, that we pass to the forward function of the hypernetwork. Otherwise, these are the "weights" passed to the forward method of the main network. Note, they might not be detached from their original parameters, because we use `backward()` on the computational graph to read out the `.grad` variable. Note, the order in which these parameters are passed to this method and the corresponding EWC loss function must not change, because the index within the "params" list will be used as unique identifier.
- **device** – Current PyTorch device.
- **mnet** – The main network. If `hnet` is `None`, then `params` are assumed to belong to this network. The fisher estimate will be computed accordingly. Note, `params` might be the output of a task-conditioned hypernetwork, i.e., weights for a specific task. In this case, "online"-EWC doesn't make much sense, as we don't follow the Bayesian view of using the old task weights as prior for the current ones. Instead, we have a new set of weights for all tasks.
- **hnet** (*optional*) – If given, `params` is assumed to correspond to the unconditional weights θ (which does not include, for instance, task embeddings) of the hypernetwork. In this case, the diagonal Fisher entries belong to weights of the hypernetwork. The Fisher will then be computed based on the probability $p(y \mid x, \text{task\_id})$, where `task_id` is just a constant input (representing the corresponding conditional weights, e.g., task embedding) in addition to the training samples x.
- **empirical_fisher** – If `True`, we compute the Fisher based on training targets.
- **online** – If `True`, then we use online EWC, hence, there is only one diagonal Fisher approximation and one target parameter value stored at the time, rather than for all previous tasks.
- **gamma** – The gamma parameter for online EWC, controlling the gradual decay of previous tasks.
- **n_max** (*optional*) – If not `-1`, this will be the maximum amount of samples considered for estimating the Fisher.

- **regression** – Whether the task at hand is a classification or regression task. If `True`, a regression task is assumed. For simplicity, we assume the following probabilistic model $p(y \mid x) = \mathcal{N}\big(f(x), I\big)$ with I being the identity matrix. In this case, the only term of the log probability that influence the gradient is the MSE: $\log p(y \mid x) = \|f(x) - y\|^2 + \text{const}$
- **time_series** (*bool*) – If `True`, the output of the main network `mnet` is expected to be a time series. In particular, we assume that the output is a tensor of shape `[S, N, F]`, where `S` is the length of the time series, `N` is the batch size and `F` is the size of each feature vector (e.g., in classification, `F` would be the number of classes).

  Let $\mathbf{y} = (\mathbf{y}_1, \dots \mathbf{y}_S)$ be the output of the main network. We denote the parameters `params` by θ and the input by $\mathbf{x}$ (which we do not consider as random). We use the following decomposition of the likelihood

$$p(\mathbf{y} \mid \theta; \mathbf{x}) = \prod_{i=1}^{S} p(\mathbf{y}_i \mid \mathbf{y}_1, \dots, \mathbf{y}_{i-1}, \theta; \mathbf{x}_i)$$

  Classification: If $f(\mathbf{x}_i, \mathbf{h}_{i-1}, \theta)$ denotes the output of the main network `mnet` for timestep i (assuming $\mathbf{h}_{i-1}$ is the most recent hidden state), we assume

$$p(\mathbf{y}_i \mid \mathbf{y}_1, \dots, \mathbf{y}_{i-1}, \theta; \mathbf{x}_i) \equiv \text{softmax}\big(f(\mathbf{x}_i, \mathbf{h}_{i-1}, \theta)\big)$$

  Hence, we assume that we can write the negative log-likelihood (NLL) as follows given a label $t \in [1, \dots, F]^S$:

$$\begin{aligned} \text{NLL} &= -\log p(Y = t \mid \theta; \mathbf{x}) \\ &= \sum_{i=1}^{S} -\text{softmax}\big(f(\mathbf{x}_i, \mathbf{h}_{i-1}, \theta)_{t_i}\big) \\ &= \sum_{i=1}^{S} \text{cross\_entropy}\big(f(\mathbf{x}_i, \mathbf{h}_{i-1}, \theta), t_i\big) \end{aligned}$$

  Thus, we simply sum the cross-entropy losses per time-step to estimate the NLL, which we then backpropagate through in order to compute the diagonal Fisher elements.
- **allowed_outputs** (*optional*) – A list of indices, indicating which output neurons of the main network should be taken into account when computing the log probability. If not specified, all output neurons are considered.
- **custom_forward** (*optional*) – A function handle that can replace the default procedure of forwarding samples through the given network(s).

  The default forward procedure if `hnet` is `None` is

```
Y = mnet.forward(X, weights=params)
```

  Otherwise, the default forward procedure is

```
weights = hnet.forward(task_id, theta=params)
Y = mnet.forward(X, weights=weights)
```

  The signature of this function should be as follows.

  - `hnet` is `None`: `@fun(mnet, params, X)`
  - `hnet` is not `None`: `@fun(mnet, hnet, task_id, params, X)`

  where `X` denotes the input batch to the main network (usually consisting of a single sample).

Example

Imagine a situation where the main network uses context- dependent modulation (cmp. `utils.context_mod_layer.ContextModLayer`) and the parameters of these context-mod layers are produced by the hypernetwork `hnet`, whereas the remaining weights of the main network `mnet` are maintained internally and passed as argument `params` to this method.

In particular, we look at a main network that is an instance of class `mnets.mlp.MLP`. The forward pass through this combination of networks should be handled as follows in order to compute the correct fisher matrix:

```
def custom_forward(mnet, hnet, task_id, params, X):
    mod_weights = hnet.forward(task_id)
    weights = {
        'mod_weights': mod_weights,
        'internal_weights': params
    }
    Y = mnet.forward(X, weights=weights)
    return Y
```

- **custom_nll** (*optional*) – A function handle that can replace the default procedure of computing the negative-log-likelihood (NLL), which is required to compute the Fisher.

  The signature of this function should be as follows:
  `@fun(Y, T, data, allowed_outputs, empirical_fisher)`

  where `Y` are the outputs of the main network. Note, `allowed_outputs` have already been applied to `Y`, if given. `T` is the target provided by the dataset `data`, transformed as follows:

  ```
  T = data.output_to_torch_tensor(batch[1], device,
                                  mode='inference')
  ```

  The arguments `data`, `allowed_outputs` and `empirical_fisher` are only passed for convinience (e.g., to apply simple sanity checks using assertions).

  The output of the function handle should be the NLL for the given sample.

- **pass_ids** (*bool*) – If a `custom_nll` is used and this flag is `True`, then the signature of the `cutom_nll` is expected to be:

  ```
  @fun(Y, T, data, allowed_outputs, empirical_fisher, batch_ids)
  ```

  where `batch_ids` are the unique identifiers as returned by option `return_ids` of method `data.dataset.Dataset.next_train_batch()` corresponding to the provided samples.

Example

In sequential datasets, target sequences `T` might be padded to the same length. Though, if the unpadded length should be used for NLL computation, then the `custom_nll` function needs the ability to request this information (sequence length) from `data`.

Also, the signatures of `custom_forward` are expected to be different.

The signature of this function should be as follows.

– `hnet` is `None`: `@fun(mnet, params, X, data, batch_ids)`

– `hnet` is not `None`: `@fun(mnet, hnet, task_id, params, X, data, batch_ids)`

- **proper_scaling** (*bool*) – The algorithm *Online EWC* is based on a Taylor approximation of the posterior that leads to the following estimate

$$\log p(\theta \mid \mathcal{D}_1, \cdots, \mathcal{D}_T) \approx \log p(\mathcal{D}_T \mid \theta) - \frac{1}{2} \sum_i \left(\sum_{t<T} N_t \mathcal{F}_{emp\ t,i} + \frac{1}{\sigma^2_{prior}} \right) (\theta_i - \theta^*_{S,i})^2 + \text{const}$$

  Due to the presentation of the algorithm in the paper and inspired by multiple publicly implementations, we approximate the regularization strength in practice via

$$\sum_{t<T} N_t \mathcal{F}_{emp\ t,i} + \frac{1}{\sigma^2_{prior}} \approx \lambda \sum_{t<T} \mathcal{F}_{emp\ t,i}$$

  where λ is a hyperparameter.

  If this argument is `True`, then the sum of Fisher matrices is properly weighted by the dataset size (independent of argument `n_max`).

- **prior_strength** (*float or list, optional*) – Either a scalar or a list of Tensors with the same shapes as `params`. Only applies to *Online EWC*. One can specify an offset for all Fisher values, e.g., $\frac{1}{\sigma^2_{prior}}$. See argument `proper_scaling` for details.

- **regression_lvar** (*float*) – In regression, this refers to the variance of the likelihood.

- **target_manipulator** (*func, optional*) – A function with signature

```
T = target_manipulator(T)
```

  That may manipulate the targets coming from the dataset.

hypnettorch.utils.ewc_regularizer.**context_mod_forward**(*mod_weights=None*)

Create a custom forward function for function *compute_fisher()*.

See argument `custom_forward` of function *compute_fisher()* for more details.

This is a helper method to quickly retrieve a function handle that manages the forward pass for a context-modulated main network.

We assume that the interface of the main network is similar to the one of `mnets.mlp.MLP.forward()`.

Parameters

mod_weights (*optional*) – If provided, it is assumed that *compute_fisher()* is called with `hnet` set to `None`. Hence, the returned function handle will have the given context-modulation pattern hard-coded. If left unspecified, it is assumed that a `hnet` is passed to *compute_fisher()* and that this `hnet` computes only the parameters of all context-mod layers.

Returns

A function handle.

hypnettorch.utils.ewc_regularizer.**ewc_regularizer**(*task_id*, *params*, *mnet*, *hnet=None*, *online=False*, *gamma=1.0*)

Compute the EWC regularizer, that can be added to the remaining loss. Note, the hyperparameter, that trades-off the regularization strength is not yet multiplied by the loss.

This loss assumes an appropriate use of the method "compute_fisher". Note, for the current task "compute_fisher" has to be called after calling this method.

If *online* is False, this method implements the loss proposed in eq. (3) in [EWC2017], except for the missing hyperparameter *lambda*.

The online EWC implementation follows eq. (8) from [OnEWC2018] (note, that lambda does not appear in this equation, but it was used in their experiments).

Parameters
(....) – See docstring of method *compute_fisher()*.

Returns
EWC regularizer.

5.5 Helper functions for training Generative Adversarial Networks

A collection of helper functions that are useful and general for GAN training, e.g., several GAN losses.

hypnettorch.utils.gan_helpers.**accuracy**(*logit_real*, *logit_fake*, *loss_choice*)

The accuracy of the discriminator.

It is computed based on the assumption that values greater than a threshold are classified as real.

Note, the accuracy measure is only well defined for the Vanilla GAN. Though, we just look at generally preferred value ranges and generalize the concept of accuracy to the other GAN formulations using the following thresholds:

- `0.5` for Vanilla GAN and Traditional LSGAN
- `0` for Pearson Chi^2 LSGAN and WGAN.

Parameters
(....) – See docstring of function *dis_loss()*.

Returns
The relative accuracy of the discriminator.

hypnettorch.utils.gan_helpers.**concat_mean_stats**(*inputs*)

Add mean statistics to discriminator input.

GANs often run into mode collapse since the discriminator sees every sample in isolation. I.e., it cannot detect whether all samples in a batch do look alike.

A simple way to allow the discriminator to have access to batch statistics is to simply concatenate the mean (across batch dimension) of all discriminator samples to each sample.

Parameters
inputs – The input batch to the discriminator.

Returns
The modified input batch.

hypnettorch.utils.gan_helpers.**dis_loss**(*logit_real*, *logit_fake*, *loss_choice*)

Compute the loss for the discriminator.

Note, only the discriminator weights should be updated using this loss.

Parameters

- **logit_real** – Outputs of the discriminator after seeing real samples.

  Note: We assume a linear output layer.

- **logit_fake** – Outputs of the discriminator after seeing fake samples.

Note: We assume a linear output layer.

- **loss_choice** (*int*) – Define what loss function is used to train the GAN. Note, the choice of loss function also influences how the output of the discriminator network if reinterpreted or squashed (either between `[0,1]` or an arbitrary real number).

  The following choices are available.

  - `0`: Vanilla GAN (Goodfellow et al., 2014). Non-saturating loss version. Note, we additionally apply one-sided label smoothing for this loss.
  - `1`: Traditional LSGAN (Mao et al., 2018). See eq. 14 of the paper. This loss corresponds to a parameter choice $a = 0$, $b = 1$ and $c = 1$.
  - `2`: Pearson Chi^2 LSGAN (Mao et al., 2018). See eq. 13. Parameter choice: $a = -1$, $b = 1$ and $c = 0$.
  - `3`: Wasserstein GAN (Arjovski et al., 2017).

Returns
The discriminator loss.

hypnettorch.utils.gan_helpers.**gen_loss**(*logit_fake*, *loss_choice*)

Compute the loss for the generator.

Parameters
(....) – See docstring of function *dis_loss()*.

Returns
The generator loss.

5.6 Hamiltonian-Monte-Carlo

The module `utils.hmc` implements the Hamiltonian-Monte-Carlo (HMC) algorithm as described in

Neal, MCMC using Hamiltonian dynamics, 2012.

The pseudocode of the algorithm is described in Figure 2 of the paper. The algorithm uses the Leapfrog algorithm to simulate the Hamiltonian dynamics in discrete time. Therefore, two crucial hyperparameters are required: the stepsize ϵ and the number of steps L. Both hyperparameters have to be chosen with care and can drastically influence the behavior of HMC. If the stepsize ϵ is too small, we don't explore the state space efficiently and waste computation. If it is too big, the numerical error from the discretization might be come too huge and the acceptance rate rather low. In addition, we want to choose L large enough to obtain good exploration, but if we set it too large we might loop back to the starting position.

The No-U-Turn-Sampler (NUTS) has been proposed to set L automatically, such that only the stepsize ϵ has to be chosen.

Hoffman et al., "The No-U-Turn Sampler: Adaptively Setting Path Lengths in Hamiltonian Monte Carlo", 2011.

This module provides implementations for both variants, basic *HMC* and *NUTS*. Multiple parallel chains can be simulated via class *MultiChainHMC*. For Bayesian Neural Networks, the helper function *nn_pot_energy()* can be used to define the potential energy.

Notation

We largely follow the notation from Neal et al.. The variable of interest, e.g., model parameters, are encoded by the position vector q. In addition, HMC requires a momentum p. The Hamiltonian $H(q, p)$ consists of two terms, the potential energy $U(q)$ and the kinetic energy $K(p) = p^T M^{-1} p/2$ with M being a symmetric, p.d. "mass" matrix.

The Hamiltonian dynamics can thus be summarized as

$$\frac{dq_i}{dt} = \frac{\partial H}{\partial p_i} = [M^{-1}p]_i$$
$$\frac{dp_i}{dt} = -\frac{\partial H}{\partial q_i} = -\frac{\partial U}{\partial q_i}$$

The Leapfrog algorithm is a way to discretize the differential equation above in a way that is reversible and volumne preserving. The algorithm has two hyperparameters: the stepsize ϵ and the number of steps L. Below, we sketch the algorithm to update momentum and position from time t to time $t + L\epsilon$.

$$p_i(t + \frac{\epsilon}{2}) = p_i(t) - \frac{\epsilon}{2} \frac{\partial U}{\partial q_i}\big(q(t)\big)$$
$$q_i(t + l\epsilon) = q_i(t + (l-1)\epsilon) + \epsilon \frac{p_i(t + (l-1)\epsilon + \epsilon/2)}{m_i} \quad \forall l = 1..L$$
$$p_i(t + l\epsilon + \frac{\epsilon}{2}) = p_i(t + (l-1)\epsilon + \frac{\epsilon}{2}) - \epsilon \frac{\partial U}{\partial q_i}\big(q(t + l\epsilon)\big) \quad \forall l = 1..L-1$$
$$p_i(t + L\epsilon) = p_i(t + (L-1)\epsilon + \frac{\epsilon}{2}) - \frac{\epsilon}{2} \frac{\partial U}{\partial q_i}\big(q(t + L\epsilon)\big)$$

We assume a diagonal mass matrix in the position update above.

hypnettorch.utils.hmc.HMC(initial_position, ...)	This class represents the basic HMC algorithm.
hypnettorch.utils.hmc.MCMC(initial_position, ...)	Implementation of the Metropolis-Hastings algorithm.
hypnettorch.utils.hmc.MultiChainHMC(...[, ...])	Wrapper for running multiple HMC chains in parallel.
hypnettorch.utils.hmc.NUTS(initial_position, ...)	HMC with No U-Turn Sampler (NUTS).
hypnettorch.utils.hmc.leapfrog(position, ...)	Implementation of the leapfrog algorithm.
hypnettorch.utils.hmc.log_prob_standard_normal_prior(...)	Log-probability density of a standard normal prior.
hypnettorch.utils.hmc.nn_pot_energy(net, ...)	The potential energy for Bayesian inference with HMC using neural networks.

class hypnettorch.utils.hmc.**HMC**(*initial_position*, *pot_energy_func*, *stepsize=0.02*, *num_steps=1*, *inv_mass=1.0*, *logger=None*, *log_interval=100*, *writer=None*, *writer_tag=''*)

Bases: object

This class represents the basic HMC algorithm.

The algorithm is implemented as outlined in Fig. 2 of Neal et al..

The potential energy should be the negative log probability density of the target distribution to sample from (up to a constant) $U(q) = -\log p(q) + \text{const.}$.

Parameters

- **initial_position** (*torch.Tensor*) – The initial position $q(0)$.

Note: The position variable should be provided as vector. The weights of a neural network can be flattend via `mnets.mnet_interface.MainNetInterface.flatten_params()`.

- **pot_energy_func** (*func*) – A function handle computing the potential energy $U(q)$ upon receiving a position q. To sample the weights of a neural network, the helper function *nn_pot_energy()* can be used. To sample via HMC from a target distribution implemented via `torch.distributions.distribution.Distribution`, one can define a function handle as in the following example.

  Example

  ```
  d = MultivariateNormal(torch.zeros(4), torch.eye(4))
  pot_energy_func = lambda q : - d.log_prob(q)
  ```

- **stepsize** (*float*) – The stepsize ϵ of the *leapfrog()* algorithm.
- **num_steps** (*int*) – The number of steps L in the *leapfrog()* algorithm.
- **inv_mass** (*float or torch.Tensor*) – The inverse "mass" matrix as required for the computation of the kinetic energy $K(p)$. See argument `inv_mass` of function *leapfrog()* for details.
- **logger** (*logging.Logger, optional*) – If provided, the progress will be logged.
- **log_interval** (*int*) – After how many states the status should be logged.
- **writer** (*tensorboardX.SummaryWriter, optional*) – A tensorboard writer. If given, useful simulation data will be logged, like the developement of the Hamiltonian.
- **writer_tag** (*str*) – Will be added to the tensorboard tags.

property acceptance_probability

The fraction of states that have been accepted.

Type
float

clear_position_trajectory(*n=None*)

Reset attribute *position_trajectory*.

This method will no affect the counter *num_states*.

Parameters
n (*int, optional*) – If provided, only the first `n` elements of *position_trajectory* are discarded (e.g., the burn-in samples).

property current_position

The latest position $q(t)$ in the chain simulated so far.

Type
torch.Tensor

property num_states

The number of states in the chain visited so far.

The counter will be increased by method *simulate_chain()*.

Type
int

property num_steps

The number of steps L in the *leapfrog()* algorithm.

You may adapt the number of steps at any point.

Type
int

property position_trajectory

A list containing all position variables (Markov states) visited so far.

New positions will be added by the method *simulate_chain()*. To decrease the memory footprint of objects in this class, the trajectory can be cleared via method *clear_position_trajectory()*.

Type
list

simulate_chain(*n*)

Simulate the next `n` states of the chain.

The new states will be appended to attribute *position_trajectory*.

Parameters
n (*int*) – Number of HMC steps to be executed.

property stepsize

The stepsize ϵ of the *leapfrog()* algorithm.

You may adapt the stepsize at any point.

Type
float

class hypnettorch.utils.hmc.**MCMC**(*initial_position*, *pot_energy_func*, *proposal_std=1.0*, *logger=None*, *log_interval=100*, *writer=None*, *writer_tag=''*)

Bases: object

Implementation of the Metropolis-Hastings algorithm.

This class implements the basic Metropolis-Hastings algorithm as, for instance, outlined here (see alg. 1).

The Metropolis-Hastings algorithm is a simple MCMC algorithm. In contrast to *HMC*, sampling is slow as positions follow a random walk. However, the algorithm does not need access to gradient information, which makes it applicable to a wider range of applications.

We use a normal distribution $\mathcal{N}(p, \sigma^2 I)$ as proposal, where p denotes the previous position (sample point). Thus, the proposal is symmetric, and cancels in the MH steps.

The potential energy is expected to be passed as negative log-probability (up to a constant), such that

$$\frac{\pi(\tilde{p}_t)}{\pi(p_{t-1})} \propto \exp\left\{U(p_{t-1}) - U(\tilde{p}_t)\right\}$$

Parameters

- **(....)** – See docstring of class *HMC*.
- **proposal_std** (*float*) – The standard deviation σ of the proposal distribution $\tilde{p}_t \sim q(p \mid p_{t-1})$.

property acceptance_probability

The fraction of states that have been accepted.

Type
float

clear_position_trajectory(*n=None*)

Reset attribute *position_trajectory*.

This method will no affect the counter *num_states*.

Parameters
n (*int, optional*) – If provided, only the first n elements of *position_trajectory* are discarded (e.g., the burn-in samples).

property current_position

The latest position $q(t)$ in the chain simulated so far.

Type
torch.Tensor

property num_states

The number of states in the chain visited so far.

The counter will be increased by method *simulate_chain()*.

Type
int

property position_trajectory

A list containing all position variables (Markov states) visited so far.

New positions will be added by the method *simulate_chain()*. To decrease the memory footprint of objects in this class, the trajectory can be cleared via method *clear_position_trajectory()*.

Type
list

property proposal_std

The std σ of the proposal distribution.

Type
float

simulate_chain(*n*)

Simulate the next n states of the chain.

The new states will be appended to attribute *position_trajectory*.

Parameters
n (*int*) – Number of MCMC steps to be executed.

class hypnettorch.utils.hmc.**MultiChainHMC**(*initial_positions*, *pot_energy_func*, *chain_type='hmc'*, ***kwargs*)

Bases: object

Wrapper for running multiple HMC chains in parallel.

Samples obtained via an MCMC sampler are highly auto-correlated for two reasons: (1) the proposal distribution is conditioned on the previous state and (2) because of rejection (consecutive states are identical). In addition, it is unclear when the chain is long enough such that sufficient exploration has been taking place and the sample (excluding initial burn-in) can be considered an i.i.d. sample from the target distribution. For this reason, it is recommended to obtain an MCMC sample by running multiple chains in parrallel, starting from varying initial postitions $q(0)$.

This class provides a simple wrapper to instantiate multiple chains from *HMC* (and its subclasses) and provides an interface to easily simulate those chains.

Parameters

- **initial_positions** (*list or tuple*) – A list of initial positions. The length of this list will determine the number of chains to be instantiated. Each element is an initial position as described for argument `initial_position` of class *HMC*.
- **pot_energy_func** (*func*) – See docstring of class *HMC*. One may also provide a list of functions. For instance, if the potential energy of a Bayesian neural network should be computed, there might be a runtime speedup if each function uses separate model instance.
- **chain_type** (*str*) – The of HMC algorithm to be used. The following options are available:
  - `'hmc'`: Each chain will be an instance of class *HMC*.
  - `'nuts'`: Each chain will be an instance of class *NUTS*.
- ****kwargs** – Keyword arguments that will be passed to the constructor when instantiating each chain. The following particularities should be noted.
  - If a `writer` object is passed, then a chain-specific identifier is added to the corresponding `writer_tag`, except if `writer` is a string. In this case, we assume `writer` corresponds to an output directory and we construct a separate object of class `tensorboardX.SummaryWriter` per chain. In the latter case, the scalars logged across chains are all shown within the same tensorboard plot and are therefore easier comparable.
  - If a `logger` object is passed, then it will only be provided to the first chain. If a logger should be passed to multiple chain instances, then a list of objects from class `logging.Logger` is required. If entries in this list are `None`, then a simple console logger is generated for these entries that displays the chain's identity when logging a message.

property avg_acceptance_probability

The average fraction of states that have been accepted across all chains.

Type
float

property chains

The list of internally managed HMC objects.

Type
list

property num_chains

The number of chains managed by this instance.

Type
int

simulate_chains(*num_states*, *num_chains=-1*, *num_parallel=1*)

Simulate the chains to gather a certain number of new positions.

This method simulates the internal chains to add `num_states` positions to each considered chain.

Parameters

- **num_states** (*int*) – Each considered chain will be simulated for this amount of HMC steps (see argument `n` of method $HMC.simulate_chain$).
- **num_chains** (*int or list*) – The number of chains to be considered. If `-1`, then all chains will be simulated for `num_states` steps. Otherwise, the `num_chains` chains with the lowest number of states so far (according to attribute *HMC.num_states*) is simulated. Alternatively, one may specify a list of chain indices (numbers between 0 and *num_chains*).

- **num_parallel** (*int*) – How many chains should be simulated in parallel. If 1, the chains are simulated consecutively (one after another).

class hypnettorch.utils.hmc.**NUTS**(*initial_position*, *pot_energy_func*, *stepsize=0.02*, *delta_max=1000.0*, *inv_mass=1.0*, *logger=None*, *log_interval=100*, *writer=None*, *writer_tag=''*)

Bases: *HMC*

HMC with No U-Turn Sampler (NUTS).

In this class, we implement the efficient version of the NUTS algorithm (see algorithm 3 in Hoffman et al.).

NUTS eliminates the need to choose the number of Leapfrog steps L. While the algorithm is more computationally expensive than basic HMC, the reduced hyperparameter effort has been shown to reduce the overall computational cost (and it requires less human intervention).

As explained in the paper, a good heuristic to set L is to choose the highest number (for given ϵ) before the trajectory loops back to the initial position q_0, e.g., when the following quantity becomes negative

$$\frac{d}{dt}\frac{1}{2}\|q - q_0\|_2^2 = \langle q - q_0, p \rangle$$

Note, this equation assumes the *mass matrix* is the identity: $M = I$.

However, this approach is in general not time reversible, therefore NUTS proposes a recursive agorithm that allows backtracing. NUTS randomly adds subtrees to a balanced binary tree and stops when any of those subtrees starts making a "U-turn" (either forward or backward in time). This tree construction is fully symmetric and therefore reversible.

Note: The NUTS paper also proposes to combine a heuristic approach to adapt the stepsize ϵ together with L (e.g., see algorithm 6 in Hoffman et al.).

Such stepsize adaptation is currently not implemented by this class!

Parameters

- **(....)** – See docstring of class *HMC*.
- **delta_max** (*float*) – The nonnegative criterion Δ_{max} from Eq. 8 of Hoffman et al., that should ensure that we stop NUTS if the energy becomes too big.

property **num_steps**

The attribute *HMC.num_steps* does not exist for class *NUTS*! Accessing this attribute will cause an error.

simulate_chain(*n*)

Simulate the next `n` states of the chain.

The new states will be appended to attribute `position_trajectory`.

Parameters

n (*int*) – Number of HMC steps to be executed.

hypnettorch.utils.hmc.**leapfrog**(*position*, *momentum*, *stepsize*, *num_steps*, *inv_mass*, *pot_energy*)

Implementation of the leapfrog algorithm.

The leapfrog algorithm updates position q and momentum p variables by simulating the Hamiltonian dynamics in discrete time for a time window of size $L\epsilon$, where L is the number of leapfrog steps `num_steps` and ϵ is the `stepsize`.

In general, one can call this method L times while setting `num_steps=1` in order to obtain the complete trajectory. However, if not necessary, we recommend setting `num_steps=L` to save the unnecessary computation of intermediate momentum variables.

Parameters

- **position** (*torch.Tensor*) – The position variable q.
- **momentum** (*torch.Tensor*) – The momentum variable p.
- **stepsize** (*float*) – The leapfrog stepsize ϵ.
- **num_steps** (*int*) – The number of leapfrog steps L.
- **inv_mass** (*float or torch.Tensor*) – The inverse mass matrix M^{-1}. Can also be provided as vector, in case of a diagonal mass matrix, or as scalar.
- **pot_energy** (*func*) – A function handle that computes the potential energy $U\big(q(t)\big)$, receiving as only input the current position variable.

Note: The function handle `pot_energy` has to be amenable to `torch.autograd`, as the momentum update requires the partial derivatives of the potential energy.

Returns

Tuple containing:

- **position** (torch.Tensor): The updated position variable.
- **momentum** (torch.Tensor): The updated momentum variable.

Return type
(tuple)

hypnettorch.utils.hmc.**log_prob_standard_normal_prior**(*position*, *mean=0.0*, *std=1.0*)

Log-probability density of a standard normal prior.

This function can be used to compute $\log p(q)$ for $p(q) = \mathcal{N}(q; \mu, I\sigma^2)$, where I denotes the identity matrix.

This function can be passed to *nn_pot_energy()* as argument `prior_log_prob_func` using, for instance:

```
lp_func = lambda q: log_prob_standard_normal_prior(q, mean=0., std=.02)
```

Parameters

- **position** (*torch.Tensor*) – The position variable q.
- **mean** (*float or torch.Tensor*) – The mean of the diagonal Gaussian prior.
- **std** (*float or torch.Tensor*) – The diagonal covariance of the Gaussian prior.

hypnettorch.utils.hmc.**nn_pot_energy**(*net*, *inputs*, *targets*, *prior_log_prob_func*, *tau_pred=1.0*, *nll_type='regression'*)

The potential energy for Bayesian inference with HMC using neural networks.

When obtaining samples from the posterior parameter distribution of a neural network via HMC, a potential energy function has to be specified that allows evaluating the negative log-posterior up to a constant. We consider a neural network with parameters W which encodes a likelihood function $p(y \mid W; x)$ for an input x. In addition,

a prior $p(W)$ needs to be specified. Given a dataset $\mathcal{D}$ consisting of `inputs` x_n and `targets` y_n, we can specify the potential energy as (note, here $q = W$)

$$\begin{aligned} U(W) &= -\log p(\mathcal{D} \mid W) - \log p(W) \\ &= -\sum_n \log p(y_n \mid W; x_n) - \log p(W) \end{aligned}$$

where the first term corresponds to the negative log-likelihood (NLL). The precise way of computing the NLL depends on which kind of likelihood interpretation is forced onto the network (cf. argument `nll_type`).

Parameters

- **net** (mnets.mnet_interface.MainNetInterface) – The considered neural network, whose parameters are W.
- **inputs** (*torch.Tensor*) – A tensor containing all the input sample points x_n in $\mathcal{D}$.
- **targets** (*torch.Tensor*) – A tensor containing all the output sample points y_n in $\mathcal{D}$.
- **prior_log_prob_func** (*func*) – Function handle that allows computing the log-probability density of the prior for a given position variate.
- **tau_pred** (*float*) – Only applies to `nll_type='regression'`. The inverse variance of the assumed Gaussian likelihood.
- **nll_type** (*str*) – The type of likelihood interpretation enforced on the network. The following options are supported:
  - `'regression'`: The network outputs the mean of a 1D normal distribution with fixed variance.

    $$\text{NLL} = \frac{1}{2\sigma_{\text{ll}}^2} \sum_{(x,y)\in\mathcal{D}} \big(f_{\text{M}}(x, W) - y\big)^2$$

    where $f_{\text{M}}(x, W)$ is the network output and $\frac{1}{\sigma_{\text{ll}}^2}$ corresponds to `tau_pred`.
  - `'classification'`: Multi-class classification with a softmax likelihood. Note, we assume the network has linear (logit) outputs

    $$\text{NLL} = \sum_{(\mathbf{x},y)\in\mathcal{D}} \Bigg(\underbrace{-\sum_{c=0}^{C-1} [c = y] \log \Big(\text{softmax}\big(f_{\text{M}}(\mathbf{x}, W)\big)_c \Big)}_{\text{cross-entropy loss with 1-hot targets}} \Bigg)$$

    where C is the number of classes and y are integer labels. We assume that the neural network $f_{\text{M}}(\mathbf{x}, W)$ outputs logits.

    Note: We assume `targets` contains integer labels and **not** 1-hot encodings for `'classification'`!

Returns

A function handle as required by constructor argument `pot_energy_func` of class *HMC*.

Return type

(func)

5.7 Hypernetwork Regularization

We summarize our own regularizers in this module. These regularizer ensure that the output of a hypernetwork don't change.

hypnettorch.utils.hnet_regularizer.**calc_fix_target_reg**(*hnet*, *task_id*, *targets=None*, *dTheta=None*, *dTembs=None*, *mnet=None*, *inds_of_out_heads=None*, *fisher_estimates=None*, *prev_theta=None*, *prev_task_embs=None*, *batch_size=None*, *reg_scaling=None*)

This regularizer simply restricts the output-mapping for previous task embeddings. I.e., for all $j <$ task_id minimize:

$$\|\text{target}_j - h(c_j, \theta + \Delta\theta)\|^2$$

where c_j is the current task embedding for task j (and we assumed that `dTheta` was passed).

Parameters

- **hnet** – The hypernetwork whose output should be regularized; has to implement the interface `hnets.hnet_interface.HyperNetInterface`.
- **task_id** (*int*) – The ID of the current task (the one that is used to compute `dTheta`).
- **targets** (*list*) – A list of outputs of the hypernetwork. Each list entry must have the output shape as returned by the `hnets.hnet_interface.HyperNetInterface.forward()` method of the `hnet`. Note, this function doesn't detach targets. If desired, that should be done before calling this function.

  Also see *get_current_targets()*.
- **dTheta** (*list, optional*) – The current direction of weight change for the internal (unconditional) weights of the hypernetwork evaluated on the task-specific loss, i.e., the weight change that would be applied to the unconditional parameters θ. This regularizer aims to modify this direction, such that the hypernet output for embeddings of previous tasks remains unaffected. Note, this function does not detach `dTheta`. It is up to the user to decide whether dTheta should be a constant vector or might depend on parameters of the hypernet.

  Also see `utils.optim_step.calc_delta_theta()`.
- **dTembs** (*list, optional*) – The current direction of weight change for the task embeddings of all tasks that have been learned already. See `dTheta` for details.
- **mnet** – Instance of the main network. Has to be provided if `inds_of_out_heads` are specified.
- **inds_of_out_heads** – (list, optional): List of lists of integers, denoting which output neurons of the main network are used for predictions of the corresponding previous tasks. This will ensure that only weights of output neurons involved in solving a task are regularized.

  If provided, the method `mnets.mnet_interface.MainNetInterface.get_output_weight_mask of the main network ``mnet`()` is used to determine which hypernetwork outputs require regularization.
- **fisher_estimates** (*list, optional*) – A list of list of tensors, containing estimates of the Fisher Information matrix for each weight tensor in the main network and each task. Note, that `len(fisher_estimates) == task_id`. The Fisher estimates are used as importance weights for single weights when computing the regularizer.

- **prev_theta** (*list, optional*) – If given, `prev_task_embs` but not `targets` has to be specified. `prev_theta` is expected to be the internal unconditional weights $theta$ prior to learning the current task. Hence, it can be used to compute the targets on the fly (which is more memory efficient (constant memory), but more computationally demanding). The computed targets will be detached from the computational graph. Independent of the current hypernet mode, the targets are computed in `eval` mode.
- **prev_task_embs** (*list, optional*) – If given, `prev_theta` but not `targets` has to be specified. `prev_task_embs` are the task embeddings (conditional parameters) of the hypernetwork. See docstring of `prev_theta` for more details.
- **batch_size** (*int, optional*) – If specified, only a random subset of previous tasks is regularized. If the given number is bigger than the number of previous tasks, all previous tasks are regularized.

  Note: A `batch_size` smaller or equal to zero will be ignored rather than throwing an error.

- **reg_scaling** (*list, optional*) – If specified, the regulariation terms for the different tasks are scaled arcording to the entries of this list.

Returns
The value of the regularizer.

hypnettorch.utils.hnet_regularizer.**flatten_and_remove_out_heads**(*mnet*, *weights*, *allowed_outputs*)

Flatten a list of target network tensors to a single vector, such that output neurons that belong to other than the current output head are dropped.

Note, this method assumes that the main network has a fully-connected output layer.

Parameters

- **mnet** – Main network instance.
- **weights** – A list of weight tensors of the main network (must adhere the corresponding weight shapes).
- **allowed_outputs** – List of integers, denoting which output neurons of the fully-connected output layer belong to the current head.

Returns
The flattened weights with those output weights not belonging to the current head being removed.

hypnettorch.utils.hnet_regularizer.**get_current_targets**(*task_id*, *hnet*)

For all $j <$ task_id, compute the output of the hypernetwork. This output will be detached from the graph before being added to the return list of this function.

Note, if these targets don't change during training, it would be more memory efficient to store the weights θ^* of the hypernetwork (which is a fixed amount of memory compared to the variable number of tasks). Though, it is more computationally expensive to recompute $h(c_j, \theta^*)$ for all $j <$ task_id everytime the target is needed.

Note, this function sets the hypernet temporarily in eval mode. No gradients are computed.

See argument `targets` of *calc_fix_target_reg()* for a use-case of this function.

Parameters

- **task_id** (*int*) – The ID of the current task.
- **hnet** – An instance of the hypernetwork before learning a new task (i.e., the hypernetwork has the weights θ^* necessary to compute the targets).

Returns
An empty list, if `task_id` is `0`. Otherwise, a list of `task_id-1` targets. These targets can be passed to the function *calc_fix_target_reg()* while training on the new task.

5.8 Helper functions for weight initialization

The module `utils.init_utils` contains helper functions that might be useful for initialization of weights. The functions are somewhat complementary to what is already provided in the PyTorch module `torch.nn.init`.

hypnettorch.utils.init_utils.**calc_fan_in_and_out**(*shapes*)

Calculate fan-in and fan-out.

Note: This function expects the shapes of an at least 2D tensor.

Parameters
shapes (*list*) – List of integers.

Returns

- **fan_in**
- **fan_out**

Return type
(tuple) Tuple containing

hypnettorch.utils.init_utils.**xavier_fan_in_**(*tensor*)

Initialize the given weight tensor with Xavier fan-in init.

Unfortunately, `torch.nn.init.xavier_uniform_()` doesn't give us the choice to use fan-in init (always uses the harmonic mean). Therefore, we provide our own implementation.

Parameters
tensor (*torch.Tensor*) – Weight tensor that will be modified (initialized) in-place.

5.9 2D-convolutional layer without weight sharing

This module implements a biologically-plausible version of a convolutional layer that does not use weight-sharing. Such a convnet is termed "locally-connected network" in:

Bartunov et al., "Assessing the Scalability of Biologically-Motivated Deep Learning Algorithms and Architectures", NeurIPS 2018.

hypnettorch.utils.local_conv2d_layer.LocalConv2dLayer(...)	Implementation of a locally-connected 2D convolutional layer.

class hypnettorch.utils.local_conv2d_layer.**LocalConv2dLayer**(*in_channels*, *out_channels*, *in_height*, *in_width*, *kernel_size*, *stride=1*, *padding=0*, *bias=True*, *no_weights=False*)

Bases: Module

Implementation of a locally-connected 2D convolutional layer.

Since this implementation of a convolutional layer doesn't use weight- sharing, it will have more parameters than a conventional convolutional layer such as torch.nn.Conv2d.

For example, consider a convolutional layer with kernel size [K, K], C_in input channels and C_out output channels, that has an output feature map size of [H, W]. Each receptive field[2] will have its own weights, a parameter tensor of size K x K. Thus, in total the layer will have C_out * C_in * H * W * K * K weights compared to C_out * C_in * K * K weights that a conventional torch.nn.Conv2d would have.

Consider the i-th input feature map $F^{(i)}$ ($1 \leq i \leq C_{\text{in}}$), the j-th output feature map $G^{(j)}$ ($1 \leq j \leq C_{\text{out}}$) and the pixel with coordinates (x, y) in the j-th output feature map $G^{(j)}_{xy}$ ($1 \leq x \leq W$ and $1 \leq y \leq H$).

We denote the filter weights of this pixel connecting to the i-th input feature map by $W^{(i,j)}_{xy} \in \mathbb{R}^{K \times K}$. The corresponding receptive field inside $F^{(i)}$ that is used to compute pixel $G^{(j)}_{xy}$ is denoted by $\hat{F}^{(i)}(x, y) \in \mathbb{R}^{K \times K}$.

The bias weights for feature map $G^{(j)}$ are denoted by $B^{(j)}$, with a scalar weight $B^{(j)}_{xy}$ for pixel (x, y).

Using this notation, the computation of this layer can be described by the following formula

$$\begin{aligned} G^{(j)}_{xy} &= B^{(j)}_{xy} + \sum_{i=1}^{C_{\text{in}}} \text{sum}(W^{(i,j)}_{xy} \odot \hat{F}^{(i)}(x, y)) \\ &= B^{(j)}_{xy} + \sum_{i=1}^{C_{\text{in}}} \langle W^{(i,j)}_{xy}, \hat{F}^{(i)}(x, y) \rangle_F \end{aligned}$$

where $\text{sum}(\cdot)$ is the unary operator that computes the sum of all elements in a matrix, $\odot$ denotes the Hadamard product and $\langle \cdot, \cdot \rangle_F$ denotes the Frobenius inner product, which computes the sum of the entries of the Hadamard product between real-valued matrices.

Implementation details

Let N denote the batch size. We can use the function torch.nn.functional.unfold() to split our input, which is of shape [N, C_in, H_in, W_in], into receptive fields F_hat of dimension [N, C_in * K * K, H * W]. The receptive field $\hat{F}^{(i)}(x, y)$ would then correspond to F_hat[:, i * K*K:(i+1) * K*K, y*H + x], assuming that indices now start at 0 and not at 1.

In addition, we have a weight tensor W of shape [C_out, C_in * K * K, H * W].

Now, we can compute the element-wise product of receptive fields and their filters by introducing a slack dimension into the shape of F_hat (i.e., [N, 1, C_in * K * K, H * W]) and by using broadcasting. F_hat * W will result into a tensor of shape [N, C_out, C_in * K * K, H * W]. By summing over the third dimension dim=2 and reshaping the output we retrieve the result of our local convolutional layer.

Parameters

- **in_channels** (*int*) – Number of channels in the input image.
- **out_channels** (*int*) – Number of channels produced by the convolution.
- **in_height** (*int*) – Height of the input feature maps, assuming that input feature maps have shape [C_in, H, W] (omitting the batch dimension). This argument is necessary to compute the size of output feature maps, as we need a filter for each pixel in each output feature map.
- **in_width** (*int*) – Width of input feature maps.

[2] For each of the C_in input feature maps, there is one receptive field for each pixel in all C_out feature maps.

- **kernel_size** (*int or tuple*) – Size of the convolving kernel.
- **stride** (*int or tuple, optional*) – Stride of the convolution.
- **padding** (*int or tuple, optional*) – Zero-padding added to both sides of the input.
- **bias** (*bool, optional*) – If `True`, adds a learnable bias to the output. There will be one scalar bias per filter.
- **no_weights** (*bool*) – If `True`, the layer will have no trainable weights. Hence, weights are expected to be passed to the *forward()* method.

Initializes internal Module state, shared by both nn.Module and ScriptModule.

forward(*x*, *weights=None*)

Compute output of local convolutional layer.

Parameters

- **x** – The input images of shape `[N, C_in, H_in, W_in]`, where `N` denotes the batch size..
- **weights** – Weights that should be used instead of the internally maintained once (determined by attribute *weights*). Note, if `no_weights` was `True` in the constructor, then this parameter is mandatory.

Returns

The output feature maps of shape `[N, C_out, H, W]`.

property out_height

Height of the output feature maps.

Type

int

property out_width

Width of the output feature maps.

Type

int

property param_shapes

A list of list of integers. Each list represents the shape of a parameter tensor. Note, this attribute is independent of the attribute *weights*, it always comprises the shapes of all weight tensors as if the network would be stand-alone (i.e., no weights being passed to the *forward()* method).

Type

list

training: bool

property weights

A list of all internal weights of this layer. If all weights are assumed to be generated externally, then this attribute will be `None`.

Type

torch.nn.ParameterList or None

5.10 Console/file logging

Collection of methods used to setup and maintain the logger used by this framework.

hypnettorch.utils.logger_config.**config_logger**(*name*, *log_file*, *file_level*, *console_level*)

Configure the logger that should be used by all modules in this package. This method sets up a logger, such that all messages are written to console and to an extra logging file. Both outputs will be the same, except that a message logged to file contains the module name, where the message comes from.

The implementation is based on an earlier implementation of a function I used in another project:

https://git.io/fNDZJ

Parameters

- **name** – The name of the created logger.
- **log_file** – Path of the log file. If None, no logfile will be generated. If the logfile already exists, it will be overwritten.
- **file_level** – Log level for logging to log file.
- **console_level** – Log level for logging to console.

Returns
The configured logger.

5.11 Miscellaneous Utilities

A collection of helper functions.

hypnettorch.utils.misc.**configure_matplotlib_params**(*fig_size=[6.4, 4.8]*, *two_axes=True*, *font_size=8*, *usetex=False*)

Helper function to configure default matplotlib parameters.

Parameters

- **fig_size** – Figure size (width, height) in inches.
- **usetex** (*bool*) – Whether `text.usetex` should be set (leads to an error on systems that don't have latex installed).

hypnettorch.utils.misc.**get_colorbrewer2_colors**(*family='Set2'*)

Helper function that returns a list of color combinations extracted from colorbrewer2.org.

Parameters
(list) – the color family from colorbrewer2.org to use.

hypnettorch.utils.misc.**get_default_args**(*func*)

Get the default values of all keyword arguments for a given function.

Parameters
func – A function handle.

Returns
Dictionary with keyword argument names as keys and their default value as values.

Return type
(dict)

hypnettorch.utils.misc.**init_params**(*weights*, *bias=None*)

Initialize the weights and biases of a linear or (transpose) conv layer.

Note, the implementation is based on the method "reset_parameters()", that defines the original PyTorch initialization for a linear or convolutional layer, resp. The implementations can be found here:

https://git.io/fhnxV

https://git.io/fhnx2

Deprecated since version 1.0: Please use function `utils.torch_utils.init_params()` instead.

Parameters

- **weights** – The weight tensor to be initialized.
- **bias** (*optional*) – The bias tensor to be initialized.

hypnettorch.utils.misc.**list_to_str**(*list_arg*, *delim=' '*)

Convert a list of numbers into a string.

Parameters

- **list_arg** – List of numbers.
- **delim** (*optional*) – Delimiter between numbers.

Returns
List converted to string.

Return type
(str)

hypnettorch.utils.misc.**repair_canvas_and_show_fig**(*fig*, *close=True*)

If writing a figure to tensorboard via "add_figure" it might change the canvas, such that our backend doesn't allow to show the figure anymore. This method will generate a new canvas and replace the old one of the given figure.

Parameters

- **fig** – The figure to be shown.
- **close** – Whether the figure should be closed after it has been shown.

hypnettorch.utils.misc.**str_to_act**(*act_str*)

Convert the name of an activation function into the actual PyTorch activation function.

Parameters
act_str – Name of activation function (as defined by command-line arguments).

Returns
Torch activation function instance or `None`, if `linear` is given.

hypnettorch.utils.misc.**str_to_floats**(*str_arg*)

Helper function to convert a string which is a list of comma separated floats into an actual list of floats.

Parameters
str_arg – String containing list of comma-separated floats. For convenience reasons, we allow the user to also pass single float that a put into a list of length 1 by this function.

Returns
List of floats.

Return type
(list)

hypnettorch.utils.misc.**str_to_ints**(*str_arg*)

Helper function to convert a string which is a list of comma separated integers into an actual list of integers.

Parameters

str_arg – String containing list of comma-separated ints. For convenience reasons, we allow the user to also pass single integers that a put into a list of length 1 by this function.

Returns

List of integers.

Return type

(list)

5.12 Compute Parameter Changes without Update Steps

PyTorch optimizers don't provide the ability to get a lookahead of the change to the parameters applied by the `torch.optim.Optimizer.step()` method. Therefore, this module copies `step()` functions from some optimizers, but without applying the weight change and without making changes to the internal state of an optimizer, such that the user can get the change of parameters that would be executed by the optimizer.

hypnettorch.utils.optim_step.**adam_step**(*optimizer*, *detach_dp=True*)

Performs a single optimization step using the Adam optimizer. The code has been copied from:

https://git.io/fjYP3

Note, this function does not change the inner state of the given optimizer object.

Note, gradients are cloned and detached by default.

Parameters

- **optimizer** – An instance of class `torch.optim.Adam`.
- **detach_dp** – Whether gradients are detached from the computational graph. Note, `False` only makes sense if func:*torch.autograd.backward* was called with the argument *create_graph* set to `True`.

Returns

A list of gradient changes *d_p* that would be applied by this optimizer to all parameters when calling `torch.optim.Adam.step()`.

hypnettorch.utils.optim_step.**calc_delta_theta**(*optimizer*, *use_sgd_change*, *lr=None*, *detach_dt=True*)

Calculate $\Delta\theta$, i.e., the change in trainable parameters (θ) in order to minimize the task-specific loss.

Note, one has to call `torch.autograd.backward()` on a desired loss before calling this function, otherwise there are no gradients to compute the weight change that the optimizer would cause. Hence, this method is called in between `torch.autograd.backward()` and `torch.optim.Optimizer.step()`.

Note, by default, gradients are detached from the computational graph.

Parameters

- **optimizer** – The optimizer that will be used to change θ.
- **use_sgd_change** – If `True`, then we won't calculate the actual step done by the current optimizer, but the one that would be done by a simple SGD optimizer.
- **lr** – Has to be specified if *use_sgd_change* is `True`. The learning rate if the optimizer.

- **detach_dt** – Whether $\Delta\theta$ should be detached from the computational graph. Note, in order to backprop through $\Delta\theta$, you have to call `torch.autograd.backward()` with *create_graph* set to `True` before calling this method.

Returns
$\Delta\theta$

hypnettorch.utils.optim_step.**rmsprop_step**(*optimizer*, *detach_dp=True*)

Performs a single optimization step using the RMSprop optimizer. The code has been copied from:

https://git.io/fjurp

Note, this function does not change the inner state of the given optimizer object.

Note, gradients are cloned and detached by default.

Parameters

- **optimizer** – An instance of class `torch.optim.Adam`.
- **detach_dp** – Whether gradients are detached from the computational graph. Note, `False` only makes sense if func:*torch.autograd.backward* was called with the argument *create_graph* set to `True`.

Returns
A list of gradient changes *d_p* that would be applied by this optimizer to all parameters when calling `torch.optim.RMSprop.step()`.

hypnettorch.utils.optim_step.**sgd_step**(*optimizer*, *detach_dp=True*)

Performs a single optimization step using the SGD optimizer. The code has been copied from:

https://git.io/fjYit

Note, this function does not change the inner state of the given optimizer object.

Note, gradients are cloned and detached by default.

Parameters

- **optimizer** – An instance of class `torch.optim.SGD`.
- **detach_dp** – Whether gradients are detached from the computational graph. Note, `False` only makes sense if func:*torch.autograd.backward* was called with the argument *create_graph* set to `True`.

Returns
A list of gradient changes *d_p* that would be applied by this optimizer to all parameters when calling `torch.optim.SGD.step()`.

5.13 Self-Attention Layer

This function was copied from

https://github.com/heykeetae/Self-Attention-GAN/blob/master/sagan_models.py

It was written by Cheonbok Park. Unfortunately, no license was visibly provided with this code.

Note, that we use this code WITHOUT ANY WARRANTIES.

The code was slightly modified to fit our purposes.

class hypnettorch.utils.self_attention_layer.**SelfAttnLayer**(*in_dim*, *use_spectral_norm*)

Bases: Module

Self-Attention Layer

This type of layer was proposed by:

> Zhang et al., "Self-Attention Generative Adversarial Networks", 2018 https://arxiv.org/abs/1805.08318

The goal is to capture global correlations in convolutional networks (such as generators and discriminators in GANs).

Initialize self-attention layer.

Parameters

- **in_dim** – Number of input channels (C).
- **use_spectral_norm** – Enable spectral normalization for all 1x1 conv. layers.

forward(*x*, *ret_attention=False*)

Compute and apply attention map to mix global information into local features.

Parameters

- **x** – Input feature maps (shape: B x C x W x H).
- **ret_attention** (*optional*) – If the attention map should be returned as an additional return value.

Returns

Tuple (if `ret_attention` is `True`) containing:

- **out**: gamma * (self-)attention features + input features.
- **attention**: Attention map, shape: B X N X N (N = W * H).

Return type
(tuple)

training: bool

class hypnettorch.utils.self_attention_layer.**SelfAttnLayerV2**(*in_dim*, *use_spectral_norm*, *no_weights=False*, *init_weights=None*)

Bases: Module

Self-Attention Layer with weights maintained separately. Hence, this class should have the exact same behavior as "SelfAttnLayer" but the weights are maintained independent of the preimplemented PyTorch modules, which allows more flexibility (e.g., generating weights by a hypernet or modifying weights easily).

This type of layer was proposed by:

> Zhang et al., "Self-Attention Generative Adversarial Networks", 2018 https://arxiv.org/abs/1805.08318

The goal is to capture global correlations in convolutional networks (such as generators and discriminators in GANs).

Initialize self-attention layer.

Parameters

- **in_dim** – Number of input channels (C).

- **use_spectral_norm** – Enable spectral normalization for all 1x1 conv. layers.
- **no_weights** – If set to True, no trainable parameters will be constructed, i.e., weights are assumed to be produced ad-hoc by a hypernetwork and passed to the forward function.
- **init_weights** (*optional*) – This option is for convinience reasons. The option expects a list of parameter values that are used to initialize the network weights. As such, it provides a convinient way of initializing a network with a weight draw produced by the hypernetwork. See attribute "weight_shapes" for the format in which parameters should be passed.

forward(*x*, *ret_attention=False*, *weights=None*, *dWeights=None*)

Compute and apply attention map to mix global information into local features.

Parameters

- **x** – Input feature maps (shape: B x C x W x H).
- **ret_attention** (*optional*) – If the attention map should be returned as an additional return value.
- **weights** – List of weight tensors, that are used as layer parameters. If "no_weights" was set in the constructor, then this parameter is mandatory. Note, when provided, internal parameters are not used.
- **dWeights** – List of weight tensors, that are added to "weights" (the internal list of parameters or the one given via the option "weights"), when computing the output of this network.

Returns

Tuple (if `ret_attention` is `True`) containing:

- **out**: gamma * (self-)attention features + input features.
- **attention**: Attention map, shape: B X N X N (N = W * H).

Return type
(tuple)

training: bool

property weight_shapes

The shapes of all parameter tensors in this layer (value of attribute is independent of whether "no_weights" was set in the constructor).

Type
list

property weights

A list of parameter tensors (all parameters in this layer). Will be `None` if this network has no weights.

Type
torch.nn.ParameterList or None

5.14 Synaptic Intelligence

The module `utils.si_regularizer` implements the Synaptic Intelligence (SI) regularizer proposed in

Zenke et al., "Continual Learning Through Synaptic Intelligence", 2017. https://arxiv.org/abs/1703.04200

Note: We aim to follow the suggested implementation from appendix section A.2.3 in

van de Ven et al., "Three scenarios for continual learning", 2019. https://arxiv.org/pdf/1904.07734.pdf

We additionally ensure that importance weights Ω are positive.

Note: This implementation has the following memory requirements. Let n denote the number of parameters to be regularized.

We always need to store the importance weights Ω and the checkpointed weights after learning the last task θ_{prev}.

We also need to checkpoint the weights right before the optimizer step is performed $\theta_{\text{pre\_step}}$ in order to update the running importance estimate ω.

Hence, we keep an additional memory of $4n$.

---

hypnettorch.utils.si_regularizer.si_pre_optim_step(...)	Prepare SI importance estimate before running the optimizer step.
hypnettorch.utils.si_regularizer.si_post_optim_step(...)	Update running importance estimate ω.
hypnettorch.utils.si_regularizer.si_compute_importance(...)	Compute weight importance Ω after training a task.
hypnettorch.utils.si_regularizer.si_regularizer(...)	Apply synaptic intelligence regularizer.

hypnettorch.utils.si_regularizer.**si_compute_importance**(*net*, *params*, *params_name=None*, *epsilon=0.001*)

Compute weight importance Ω after training a task.

Note: This function is assumed to be called after the training on the current task finished. It will set the variable θ_{prev} to the current parameter value.

Parameters

- **(....)** – See docstring of function *si_pre_optim_step()*.
- **epsilon** (*float*) – Damping parameter used to ensure numerical stability when normalizing weight importance.

hypnettorch.utils.si_regularizer.**si_post_optim_step**(*net*, *params*, *params_name=None*, *delta_params=None*)

Update running importance estimate ω.

This function is called after an optimizer update step has been performed. It will perform an update of the internal running variable :math:omega` using the current parameter values, the checkpointed parameter values before the

optimizer step ($\theta_{\text{pre\_step}}$, see function *si_pre_optim_step()*) and the negative gradients accumulated in the `grad` variables of the parameters.

Parameters

- **(....)** – See docstring of function *si_pre_optim_step()*.
- **delta_params** (*list*) – One may pass the parameter update step directly. In this case. the difference between the current parameter values and the previous ones $\theta_{\text{pre\_step}}$ will not be computed.

Note: One may use the functions provided in module `utils.optim_step` to calculate `delta_params`

Note: When this option is used, it is not required to explicitly call the optimizer its `step` function. Though, it is still required that gradients are computed and accumulated in the `grad` variables of the parameters in `params`.

Note: This option is particularly interesting if importances should only be estimated wrt to a part of the total loss function, e.g., the task-specific part, ignoring other parts of the loss (e.g., regularizers).

---

hypnettorch.utils.si_regularizer.**si_pre_optim_step**(*net*, *params*, *params_name=None*, *no_pre_step_ckpt=False*)

Prepare SI importance estimate before running the optimizer step.

This function has to be called before running the optimizer step in order to checkpoint $\theta_{\text{pre\_step}}$.

Note: When this function is called the first time (for the first task), the given parameters will also be checkpointed as the initial weights, which are required to normalize importances :math:Omega` after training.

Parameters

- **net** (*torch.nn.Module*) – A network required to store buffers (i.e., the running variables that SI needs to keep track of).
- **params** (*list*) – A list of parameter tensors. For each parameter tensor in this list that `requires_grad` the importances will be measured.
- **params_name** (*str, optional*) – In case SI should be performed for multiple parameter groups `params`, one has to assign names to each group via this option.
- **no_pre_step_ckpt** (*bool*) – If `True`, then this function will not checkpoint $\theta_{\text{pre\_step}}$. Instead, option `delta_params` of function *si_post_optim_step()* is expected to be set.

Note: One still has to call this function once before updating the parameters of the first task for the first time.

---

hypnettorch.utils.si_regularizer.**si_regularizer**(*net*, *params*, *params_name=None*)

Apply synaptic intelligence regularizer.

This function computes the SI regularizer. Note, a regularization strength should be multiplied by the returned loss post-hoc, to tune the strength.

Parameters
(....) – See docstring of function *si_pre_optim_step()*.

Returns
The regularizer as scalar value.

Return type
(torch.Tensor)

5.15 General helper functions for simulations

The module `utils.sim_utils` comprises a bunch of functions that are in general useful for writing simulations in this repository.

hypnettorch.utils.sim_utils.**calc_train_iter**(*num_train_samples*, *batch_size*, *num_iter=-1*, *epochs=-1*)

Calculate the number of training tierations.

If `epochs` is specified, this method will compute the total number of training iterations and the number of iterations per epoch.

Otherwise, the number of training iterations is simply set to `num_iter`.

Parameters

- **num_train_samples** (*int*) – Numbe rof training samples in dataset.
- **batch_size** (*int*) – Mini-batch size during training.
- **num_iter** (*int*) – Number of training iterations. Only needs to be specified if `epochs` is `-1`.
- **epochs** (*int, optional*) – Number of training epochs.

Returns

Tuple containing:

- **num_train_iter**: Total number of training iterations.
- **iter_per_epoch**: Number of training iterations per epoch. Is set to `-1` in case `epochs` is unspecified.

Return type
(tuple)

hypnettorch.utils.sim_utils.**get_hypernet**(*config*, *device*, *net_type*, *target_shapes*, *num_conds*, *no_cond_weights=False*, *no_uncond_weights=False*, *uncond_in_size=0*, *shmlp_chunk_shapes=None*, *shmlp_num_per_chunk=None*, *shmlp_assembly_fct=None*, *verbose=True*, *cprefix=None*)

Generate a hypernetwork instance.

A helper to generate the hypernetwork according to the given the user configurations.

Parameters

- **config** (*argparse.Namespace*) – Command-line arguments.

  Note: The function expects command-line arguments available according to the function `utils.cli_args.hnet_args()`.

- **device** – PyTorch device.
- **net_type** (*str*) – The type of network. The following options are available:
  - `'hmlp'`
  - `'chunked_hmlp'`
  - `'structured_hmlp'`
  - `'hdeconv'`
  - `'chunked_hdeconv'`
- **target_shapes** (*list*) – See argument `target_shapes` of `hnets.mlp_hnet.HMLP`.
- **num_conds** (*int*) – Number of conditions that should be known to the hypernetwork.
- **no_cond_weights** (*bool*) – See argument `no_cond_weights` of `hnets.mlp_hnet.HMLP`.
- **no_uncond_weights** (*bool*) – See argument `no_uncond_weights` of `hnets.mlp_hnet.HMLP`.
- **uncond_in_size** (*int*) – See argument `uncond_in_size` of `hnets.mlp_hnet.HMLP`.
- **shmlp_chunk_shapes** (*list, optional*) – Argument `chunk_shapes` of `hnets.structured_mlp_hnet.StructuredHMLP`.
- **shmlp_num_per_chunk** (*list, optional*) – Argument `num_per_chunk` of `hnets.structured_mlp_hnet.StructuredHMLP`.
- **shmlp_assembly_fct** (*func, optional*) – Argument `assembly_fct` of `hnets.structured_mlp_hnet.StructuredHMLP`.
- **verbose** (*bool*) – Argument `verbose` of `hnets.mlp_hnet.HMLP`.
- **cprefix** (*str, optional*) – A prefix of the config names. It might be, that the config names used in this function are prefixed, since several hypernetworks should be generated.

  Also see docstring of parameter `prefix` in function `utils.cli_args.hnet_args()`.

hypnettorch.utils.sim_utils.**get_mnet_model**(*config*, *net_type*, *in_shape*, *out_shape*, *device*, *cprefix=None*, *no_weights=False*, ***mnet_kwargs*)

Generate a main network instance.

A helper to generate a main network according to the given the user configurations.

Note: Generation of networks with context-modulation is not yet supported, since there is no global argument set in `utils.cli_args` yet.

Parameters

- **config** (*argparse.Namespace*) – Command-line arguments.

Note: The function expects command-line arguments available according to the function `utils.cli_args.main_net_args()`.

- **net_type** (*str*) – The type of network. The following options are available:
  - `mlp`: `mnets.mlp.MLP`
  - `resnet`: `mnets.resnet.ResNet`
  - `wrn`: `mnets.wide_resnet.WRN`
  - `iresnet`: `mnets.resnet_imgnet.ResNetIN`
  - `zenke`: `mnets.zenkenet.ZenkeNet`
  - `bio_conv_net`: `mnets.bio_conv_net.BioConvNet`
  - `chunked_mlp`: `mnets.chunk_squeezer.ChunkSqueezer`
  - `simple_rnn`: `mnets.simple_rnn.SimpleRNN`
- **in_shape** (*list*) – Shape of network inputs. Can be `None` if not required by network type.

  For instance: For an MLP network `mnets.mlp.MLP` with 100 input neurons it should be `in_shape=[100]`.
- **out_shape** (*list*) – Shape of network outputs. See `in_shape` for more details.
- **device** – PyTorch device.
- **cprefix** (*str, optional*) – A prefix of the config names. It might be, that the config names used in this method are prefixed, since several main networks should be generated (e.g., `cprefix='gen_'` or `'dis_'` when training a GAN).

  Also see docstring of parameter `prefix` in function `utils.cli_args.main_net_args()`.
- **no_weights** (*bool*) – Whether the main network should be generated without weights.
- ****mnet_kwargs** – Additional keyword arguments that will be passed to the main network constructor.

Returns
The created main network model.

hypnettorch.utils.sim_utils.**setup_environment**(*config*, *logger_name='hnet_sim_logger'*)

Setup the general environment for training.

This function should be called at the beginning of a simulation script (right after the command-line arguments have been parsed). The setup will incorporate:

- creating the output folder
- initializing logger
- making computation deterministic (depending on config)
- selecting the torch device
- creating the Tensorboard writer

Parameters

- **config** (*argparse.Namespace*) – Command-line arguments.

  Note: The function expects command-line arguments available according to the function `utils.cli_args.miscellaneous_args()`.

- **logger_name** (*str*) – Name of the logger to be created (time stamp will be appended to this name).

Returns

Tuple containing:

- **device**: Torch device to be used.
- **writer**: Tensorboard writer. Note, you still have to close the writer manually!
- **logger**: Console (and file) logger.

Return type
(tuple)

5.16 Checkpointing PyTorch Models

This module provides functions to handle PyTorch checkpoints with a similar convenience as one might be used to in Tensorflow.

hypnettorch.utils.torch_ckpts.get_best_ckpt_path(...)	Returns the path to the checkpoint with the highest score.
hypnettorch.utils.torch_ckpts.load_checkpoint(...)	Load a checkpoint from file.
hypnettorch.utils.torch_ckpts.make_ckpt_list(...)	Creates a file that lists all checkpoints together with there scores, such that one can easily find the checkpoint associated with the maximum score.
hypnettorch.utils.torch_ckpts.save_checkpoint(...)	Save checkpoint to file.

hypnettorch.utils.torch_ckpts.**get_best_ckpt_path**(*file_path*)

Returns the path to the checkpoint with the highest score.

Parameters
file_path – See method `save_checkpoints()`.

hypnettorch.utils.torch_ckpts.**load_checkpoint**(*ckpt_path*, *net*, *device=None*, *ret_performance_score=False*)

Load a checkpoint from file.

Parameters

- **ckpt_path** – Path to checkpoint.
- **net** – The network, that should load the state dict saved in this checkpoint.
- **device** (*optional*) – The device currently used by the model. Can help to speed up loading the checkpoint.

- **ret_performance_score** – If True, the score associated with this checkpoint will be returned as well. See argument "performance_score" of method "save_ckecpoint".

Returns

The loaded checkpoint. Note, the state_dict is already applied to the network. However, there might be other important dict elements.

hypnettorch.utils.torch_ckpts.**make_ckpt_list**(*file_path*)

Creates a file that lists all checkpoints together with there scores, such that one can easily find the checkpoint associated with the maximum score.

Parameters

file_path – See method `save_checkpoints()`.

hypnettorch.utils.torch_ckpts.**save_checkpoint**(*ckpt_dict*, *file_path*, *performance_score*, *train_iter=None*, *max_ckpts_to_keep=5*, *keep_cktp_every=2*, *timestamp=None*)

Save checkpoint to file.

Example

```
save_checkpoint({
    'state_dict': net.state_dict(),
    'train_iter': curr_iteration
}, 'ckpts/my_net', current_test_accuracy)
```

Parameters

- **ckpt_dict** – A dict with mostly arbitrary content. Though, most important, it needs to include the state dict and should also include the current training iteration.
- **file_path** –

  Where to store the checkpoint. Note, the filepath should
  not change. Instead, `train_iter` should be provided, such that this method can handle the filenames by itself.

  Note: The function currently assumes that within the same directory, no checkpoint filename is the prefix of another checkpoint filename (e.g., if several networks are checkpointed into the same directory).

- **performance_score** – A score that expresses the performance of the current network state, e.g., accuracy for a classification task. This score is used to maintain the list of kept checkpoints during training.
- **train_iter** (*optional*) – If given, it will be added to the filename. Otherwise, existing checkpoints are simply overwritten.
- **max_ckpts_to_keep** – The maximum number of checkpoints to keep. This will use the performance score to determine the n-1 checkpoints not to be deleted (where n is the number of checkpoints to keep). The current checkpoint will always be saved.
- **keep_cktp_every** – If this option is not `None`, then every n hours one checkpoint will be permanently saved, i.e., this checkpoint will not be maintained by 'max_ckpts_to_keep' anymore. The checkpoint to be kept will be the best one from the time window that spans the last n hours.

- **timestamp** (*optional*) – The timestamp of this checkpoint. If not given, a current timestamp will be used. This option is useful when one aims to synchronize checkpoint savings from multiple networks.

A collection of helper functions that should capture common functionalities needed when working with PyTorch.

class hypnettorch.utils.torch_utils.**CutoutTransform**(*n_holes*, *length*)

Bases: object

Randomly mask out one or more patches from an image.

The cutout transformation as preprocessing step has been proposed by

DeVries et al., Improved Regularization of Convolutional Neural Networks with Cutout, 2017.

The original implementation can be found here.

Parameters

- **n_holes** (*int*) – Number of patches to cut out of each image.
- **length** (*int*) – The length (in pixels) of each square patch.

hypnettorch.utils.torch_utils.**get_optimizer**(*params*, *lr*, *momentum=0*, *weight_decay=0*, *use_adam=False*, *adam_beta1=0.9*, *use_rmsprop=False*, *use_adadelta=False*, *use_adagrad=False*, *pgroup_ids=None*)

Create an optimizer instance for the given set of parameters. Default optimizer is torch.optim.SGD.

Parameters

- **params** (*list*) – The parameters passed to the optimizer.
- **lr** – Learning rate.
- **momentum** (*optional*) – Momentum (only applicable to torch.optim.SGD and torch.optim.RMSprop.
- **weight_decay** (*optional*) – L2 penalty.
- **use_adam** – Use torch.optim.Adam optimizer.
- **adam_beta1** – First parameter in the *betas* tuple that is passed to the optimizer torch.optim.Adam: `betas=(adam_beta1, 0.999)`.
- **use_rmsprop** – Use torch.optim.RMSprop optimizer.
- **use_adadelta** – Use torch.optim.Adadelta optimizer.
- **use_adagrad** – Use torch.optim.Adagrad optimizer.
- **pgroup_ids** (*list, optional*) – If passed, a list of integers of the same length as params is expected. In this case, each integer states to which parameter group the corresponding parameter in `params` shall belong. Parameter groups may have different optimizer settings. Therefore, options like `lr`, `momentum`, `weight_decay`, `adam_beta1` may be lists in this case that have a length corresponding to the number of parameter groups.

Returns

Optimizer instance.

hypnettorch.utils.torch_utils.**init_params**(*weights*, *bias=None*)

Initialize the weights and biases of a linear or (transpose) conv layer.

Note, the implementation is based on the method "reset_parameters()", that defines the original PyTorch initialization for a linear or convolutional layer, resp. The implementations can be found here:

https://git.io/fhnxV

https://git.io/fhnx2

Parameters

- **weights** – The weight tensor to be initialized.
- **bias** (*optional*) – The bias tensor to be initialized.

hypnettorch.utils.torch_utils.**lambda_lr_schedule**(*epoch*)

Multiplicative Factor for Learning Rate Schedule.

Computes a multiplicative factor for the initial learning rate based on the current epoch. This method can be used as argument `lr_lambda` of class `torch.optim.lr_scheduler.LambdaLR`.

The schedule is inspired by the Resnet CIFAR-10 schedule suggested here https://keras.io/examples/cifar10_resnet/.

Parameters
epoch (*int*) – The number of epochs

Returns
learning rate scale

Return type
lr_scale (float32)

CHAPTER

SIX

TUTORIALS ON HOW TO USE HYPERNETWORKS IN PYTORCH

Here, we present a series of tutorials covering different aspects of the repository `hypnettorch`. These tutorials are meant as an easy entrance point for coding with this package.

- Getting started
- How to smartly chunk the weights of a Resnet
- How to smartly chunk the weights of a Wide-Resnet
- MCMC sampling

CHAPTER

SEVEN

EXAMPLE IMPLEMENTATIONS THAT USE `HYPNETTORCH`

Contents

Let's dive into some example implementations that make use of the functionalities provided by the package `hypnettorch`. You can explore the corresponding source code to see how to efficiently make use of all the functionalities that `hypnettorch` offers.

7.1 Continual learning with hypernetworks

In continual learning (CL), a series of tasks (represented as datasets) $\mathcal{D}_1, ..., \mathcal{D}_T$ is learned sequentially, where only one dataset at a time is available and at the end of training performance on all tasks should be high.

An approach based on hypernets for tackling this problem was introduced by von Oswald, Henning, Sacramento et al.. The official implementation can be found here. Goal of this example is it to demonstrate how `hypnettorch` can be used to implement such CL approach. Therefore, we provide a simple and light implementation that showcases many functionalities inherent to the package, **but do not focus on being able to reproduce the variety of experiments explored in the original paper**.

For the sake of simplicity, we only focus on the simplest CL scenario, called task-incremental CL or `CL1` (note, that the original paper proposes three ways of tackling more complex CL scenarios, one of which has been further studied in this paper). Predictions according to a task t are made by inputting the corresponding task embedding $\mathbf{e}^{(t)}$ into the hypernetwork in order to obtain the main network's weights $\omega^{(t)} = h(\mathbf{e}^{(t)}, \theta)$, which in turn can be used for processing inputs via $f(x, \omega^{(t)})$. Forgetting is prevented by adding a simple regularizer to the loss while learning task t:

$$\frac{\beta}{t-1} \sum_{t<t'} \| h(\mathbf{e}^{(t')}, \theta) - h(\mathbf{e}^{(t',*)}, \theta^{(*)}) \|_2^2 \tag{7.1}$$

where β is a regularization constant, $\mathbf{e}^{(t')}$ are the task-embeddings, θ are the hypernets' parameters and parameters denoted by $^{(*)}$ are checkpointed from before starting to learn task t. Simply speaking, the regularizer aims to prevent that the hypernetwork output $h(\mathbf{e}^{(t')}, \theta)$ for a previous task t' changes compared to what was outputted before we started to learn task t.

Note: The original paper uses a lookahead in the regularizer which showed marginal performance improvements. Follow-up work (e.g., here and here) discarded this lookahead for computational convenience. We ignore it as well!

7.1.1 Usage instructions

The script *hypnettorch.examples.hypercl.run* showcases how a versatile simulation can be build with relatively little coding effort. You can explore the basic functionality of the script via

```
$ python run.py --help
```

Note: The default arguments have **not** been hyperparameter-searched and may thus not reflect best possible performance.

By default, the script will run a **SplitMNIST** simulation (argument `--cl_exp`)

```
$ python run.py
```

The default network (argument `--net_type`) is a 2-hidden-layer MLP and the corresponding hypernetwork has been chosen to have roughly the same number of parameters (compression ratio is approx. 1).

Via the argument `--hnet_reg_batch_size` you can choose up to how many task should be used for the regularization in Eq. (7.1) (rather than always evaluating the sum over all previous tasks). This ensures that the computational budget of the regularization doesn't grow with the number of tasks. For instance, if at every iteration a **single** random (previous) task should be selected for regularization, just use

```
$ python run.py --hnet_reg_batch_size=1
```

You can also run other CL experiments, such as **PermutedMNIST** (e.g., via arguments `--cl_exp=permmnist --num_classes_per_task=10 --num_tasks=10`) or **SplitCIFAR-10/100** (e.g., via arguments `--cl_exp=splitcifar --num_classes_per_task=10 --num_tasks=6 --net_type=resnet`). Keep in mind, that with a change in dataset or main network, model sizes change and thus another hypernetwork should be chosen if a certain compression ratio should be accomplished.

7.1.2 Learning from the example

Goal of this example is it to get familiar with the capabilities of the package `hypnettorch`. This can best be accomplished by reading through the source code, starting with the main function *hypnettorch.examples.hypercl.run.run()*.

1. The script makes use of module *hypnettorch.utils.cli_args* for defining command-line arguments. With a few lines of code, a large variety of arguments are created to, for instance, flexibly determine the architecture of the main- and hypernetwork.
2. Using those predefined arguments allows to quickly instantiate the corresponding networks by using functions of module *hypnettorch.utils.sim_utils*.
3. Continual learning datasets are generated with the help of specialized data handlers, e.g., *hypnettorch.data.special.split_mnist.get_split_mnist_handlers()*.
4. Hypernet regularization (Eq. (7.1)) is easily realized via the helper functions in module *hypnettorch.utils.hnet_regularizer*.

There are many other utilities that might be useful, but that are not incorporated in the example for the sake of simplicity. For instance:

- The module *hypnettorch.utils.torch_ckpts* can be used to easily save and load networks.
- The script can be emebedded into the hyperparameter-search framework of subpackage *hpsearch* to easily scan for hyperparameters that yield good performance.

More sophisticated examples can also be explored in the PR-CL repository (note, the interface used in this repository is almost identical to `hypnettorch`'s interface, except that the package wasn't called `hypnettorch` back then yet).

Script to run CL experiments with hypernetworks

This script showcases the usage of `hypnettorch` by demonstrating how to use the pacakge for writing a continual learning simulation that utilizes hypernetworks. See *here* for details on the approach and usage instructions.

hypnettorch.examples.hypercl.run.**evaluate**(*task_id*, *data*, *mnet*, *hnet*, *device*, *config*, *logger*, *writer*, *train_iter*)

Evaluate the network.

Evaluate the performance of the network on a single task on the validation set during training.

Parameters

(....) – See docstring of function *train()*.

train_iter (int): The current training iteration.

hypnettorch.examples.hypercl.run.**load_datasets**(*config*, *logger*, *writer*)

Load the datasets corresponding to individual tasks.

Parameters

- **config** (*argparse.Namespace*) – Command-line arguments.
- **logger** (*logging.Logger*) – Logger object.
- **writer** (*tensorboardX.SummaryWriter*) – Tensorboard logger.

Returns

A list of data handlers *hypnettorch.data.dataset.Dataset*.

Return type

(list)

hypnettorch.examples.hypercl.run.**run**()

Run the script.

1. Define and parse command-line arguments
2. Setup environment
3. Load data
4. Instantiate models
5. Run training for each task

hypnettorch.examples.hypercl.run.**test**(*dhandlers*, *mnet*, *hnet*, *device*, *config*, *logger*, *writer*)

Evaluate the network.

Evaluate the performance of the network on a single task on the validation set during training.

Parameters

- **(....)** – See docstring of function *train()*.
- **dhandlers** (*list*) – Datasets of tasks that should be tested. We assume that the index of the dataset corresponds to the index of the task embedding used as input to the hypernet.

hypnettorch.examples.hypercl.run.**train**(*task_id*, *data*, *mnet*, *hnet*, *device*, *config*, *logger*, *writer*)

Train the network using the task-specific loss plus a regularizer that should mitigate catastrophic forgetting.

$$\text{loss} = \text{task\_loss} + \beta * \text{regularizer}$$

Parameters

- **task_id** (*int*) – The index of the task on which we train.
- **data** (hypnettorch.data.dataset.Dataset) – The dataset handler for the current task, corresponding to task_id.
- **mnet** (hypnettorch.mnets.mnet_interface.MainNetInterface) – The model of the main network, which is needed to make predictions.
- **hnet** (hypnettorch.hnets.hnet_interface.HyperNetInterface) – The model of the hyper network, which contains the parameters to be learned.
- **device** – (torch.device) Torch device (cpu or gpu).
- **config** (*argparse.Namespace*) – Command-line arguments.
- **logger** (*logging.Logger*) – Logger object.
- **writer** (*tensorboardX.SummaryWriter*) – Tensorboard logger.

This package provides functionalities to easily work with hypernetworks in PyTorch. A hypernetwork $h(\mathbf{e}, \theta)$ is a neural network with parameters θ that generates the parameters ω of another neural network $f(\mathbf{x}, \omega)$, called *main network*. These two network types require specialized implementations. For instance, a *main network* must have the ability to receive its own weights ω as additional input to the forward method (see subpackage *mnets*). A collection of different hypernetwork implementations can be found in subpackage *hnets*.

CHAPTER

EIGHT

INSTALLATION

See here.

CHAPTER

NINE

USAGE

Check out the *tutorials*, especially the getting started tutorial.

You can also check out *example implementations* that make use of `hypnettorch`.

CHAPTER

TEN

INDICES AND TABLES

- genindex
- modindex
- search

BIBLIOGRAPHY

[EWC2017] https://arxiv.org/abs/1612.00796

[OnEWC2018] https://arxiv.org/abs/1805.06370

PYTHON MODULE INDEX

h

INDEX

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W

X

Z