modelhub Documentation

Release 0.4.0

Ahmed Hosny, Michael Schwier

Jun 30, 2020

Contents

Crowdsourced through contributions by the scientific research community, modelhub is a repository of deep learning models pretrained for a wide variety of medical applications. Modelhub highlights recent trends in deep learning applications, enables transfer learning approaches and promotes reproducible science.

Note: This documentation should contain all essential technical information about the Modelhub project and how to contribute models. It is, however, still work-in-progress, so possibly you need to be a little patient and persistent. If you find anything unclear, need help, or have suggestions, please feel free to contact us at “info at modelhub.ai”

CHAPTER 1

Contents:

1.1 Quick Start

The most accessible way to experience modelhub is via modelhub.ai. There you can explore the model collection, try them online, and find instructions on how to run models locally.

But since you are here, follow these steps to get modelhub running on your local computer:

1. **Install Docker** (if not already installed)

   Follow the official Docker instructions to install Docker CE. Docker is required to run models. **GPU Support**: If you want to run models that require GPU acceleration, please use Docker version >= 19.03 and follow the installation instructions for the Nvidia-Docker Toolkit here.

2. **Install Python 2.7 or 3.6 (or higher)** (if not already installed)

   Download and install Python from the official Python page. Modelhub requires Python 2.7 or Python 3.6 (or higher).

3. **Install the modelhub-ai package**

   Install the `modelhub-ai` package from PyPi using pip: `pip install modelhub-ai`.

4. **Run a model using start.py**

   Open a terminal and navigate to a folder you want to work in. For running models, write access is required in the current folder.

   Execute `modelhub-run squeezenet` in the terminal to run the squeezenet model from the modelhub collection. This will download all required model files (only if they do not exist yet) and start the model. Follow the instructions given on the terminal to access the web interface to explore the model.

   Replace `squeezenet` by any other model name in the collection to start a different model. To see a list of all available models execute `modelhub-list` or `modelhub -l`.

   You can also access a jupyter notebook that allows you to experiment with a model by starting a model with the "-e" option, e.g. `modelhub-run squeezenet -e`. Follow the instructions on the terminal to open the notebook.

See additional starting options by executing `modelhub-run -h`.

1.2 Overview

1.2.1 Framework

Modelhub provides a framework into which contributors can plug-in their model, and model specific pre- and post-processing code. The framework provides a standalone runtime environment, convience functionality (e.g. image loading and conversion), programming interfaces to access the model, and a user friendly web-interface to try a model. See the following figure for an overview of the architecture.

The *contrib_src* contains the model specific code and data, all other functionality is provided by the framework. The framework and model specific code run inside of a Docker container, which contains all runtime dependencies. The resulting package constitutes a standalone unit that can be easily deployed, executed on different platforms (Linux, Windows, Mac), and integrated into existing applications via the generic API.

1.2.2 Repository Structure

The whole modelhub infrastructure is a combination of several repositories under https://github.com/modelhub-ai, comprising the following:

- **modelhub** Index/Registry of all models

  Contains

  - a list (index/registry) of all models available via modelhub
  - json schema for validating model config files
  - python script to conveniently start any model which is registered in the modelhub index

- **modelhub-app** Generic web frontend for a model

  Web app for easy interaction with a model provides

  - relevant info about model (architecture, I/O, purpose)
  - info about accompanying publication (optional)
  - GUI interface to run/test the model

  The web app is generic and works on top of every model without modifications.

- **modelhub-engine** Backend library, framework, and API

  Library and common framework on which model contributors must base their model contribution. The framework handles/provides -data I/O -data conversion to/from numpy (typical data format used in deep learning libraries) -generic API for accessing and working with the model -"slots" for preprocessing, postprocessing, and inference, which have to be populated by the contributor with the model specific code

- **model-template** Template structure for building modelhub compatible models

  Defines the file and directory structure required to build a model that can be integrated into modelhub. Contributors should clone this repository and build fill in the template with their model specific code/info.

- **<model name>** A model implementation available via modelhub

  Several models are directly hosted under modelhub.ai. Each model has its own repository. The structure of the repository follows the model-template. However, models don't need to be hosted under modelhub.ai but can be any github repository. To be integrated in and available via modelhub, they only have to be listed in the modelhub index/registry.

- **modelhub-ai.github.io** Modelhub webpage

  Source code for the modelhub.ai webpage

1.3 Contribute Your Model to Modelhub

The following figure gives an overview of the necessary steps to package your model with the Modelhub framework and eventually contributing it to the Modelhub collection. Read further for detailed explanations of all steps.

HINT Take a look at an already integrated model to understand how it looks when finished (AlexNet is a good and simple example. If you have a more complex model with more than one input for a single inference, have a look at one of the BraTS models, e.g. lfb-rwth).

1.3.1 Prerequisites

To package a model with our framework you need to have the following prerequisites installed:

- Python 2.7 or Python 3.6 (or higher)
- Docker
- Clone of the modelhub-engine repository (`git clone https://github.com/modelhub-ai/modelhub-engine.git`)
- For GPU support, you need Docker version >= 19.03 and follow the instructions here.

1.3.2 1. Prepare Docker image

1. Write a dockerfile preparing/installing all third party dependencies your model needs (e.g. the deep learning library you are using). Use the `ubuntu:16.04` Docker image as base. If you want to use CUDA and GPU acceleration, you can also use one of the `nvidia/cuda` images as base.

   You can find examples of dockerfiles for DL environments in the model repositories of modelhub-ai on github (e.g. for squeezenet).

2. Build the docker image.
3. Adapt the *Dockerfile_modelhub* located in the modelhub-engine repository to use your docker image as base (i.e. change the `FROM XXXXXXXX` line to `FROM <your docker image>`). No other changes should be necessary.
4. Build the image from the modified *Dockerfile_modelhub*. This will include the modelhub engine into your docker. Make sure to build it from within the modelhub-engine repository so it finds the modelhub framework which it will include in the Docker.
5. Push the image from the previous step to DockerHub (required if you want to publish your model on Modelhub, so the image can be found when starting a model for the first time. If you don't plan to publish on Modelhub, this step is optional).

- ***NOTE*** We are planning to provide a few pre-built Docker images for the most common deep learning frameworks, so you do not have to build them yourself. For now we only have a small set. You can find the existing pre-build images on DockerHub - use the ones that end with '-modelhub' (the ones that don't end with '-modelhub' have only the pure DL environment without the modelhub framework on top.

If the DL environment, the exact version of the DL environment, or third party dependencies you require are not available in the pre-build dockers, you have to build it yourself, following the above steps.

1.3.3 2. Prepare your model based on the modelhub template

1. Fork the model template.
2. Change the name of your model-template fork to your model's name. For this open your fork on GitHub, go to *Settings*, change the *Repository name*, and press *Rename*.
3. Clone your renamed fork to your local computer and open the cloned folder.
4. Populate the configuration file *contrib_src/model/config.json* with the relevant information about your model. Please refer to the schema for allowed values and structure.

   Version 0.4 and up breaks the compatibility with older versions of the schema, please validate your configuration file against the current schema if you are submitting a new model. Old models are still compatible anddon't need to be changed unless you are updating the modelhub-engine version of the Docker image. For single-input models, assign the key `"single"` to your input as in the schema above. ***HINT*** For more details on how to set up your model for various input scenarios and implement your own ImageLoader class, see the IO Configuration documentation.
5. Place your pre-trained model file(s) into the *contrib_src/model/* folder.
6. (optional) Place some sample data into the *contrib_src/sample_data/* folder. This is not mandatory but highly recommended, so users can try your model directly.
7. Open *contrib_src/inference.py* and replace the model initialization and inference with your model specific code. The template example shows how to integrate models in ONNX format and running them in caffe2. If you are using a different model format and/or backend you have to change this.

   There are only two lines you have to modify. In the `__init__` function change the following line, which loads the model:

   ```
   # load the DL model (change this if you are not using ONNX)
   self._model = onnx.load('model/model.onnx')
   ```

   If your model receives more than one file as input, the `input` argument of `infer` is a dictionary matching the input schema specified in `config.json`. You would then need to pass each individual input through the preprocessing and to your inference function. For example, accessing the input `image_pose` would look like this: `input["image_pose"]["fileurl"]`.

   In the `infer` function change the following line, which runs the model prediction on the input data:

   ```
   # Run inference with caffe2 (change this if you are using a different DL␣
   ↪framework)
   results = caffe2.python.onnx.backend.run_model(self._model, [inputAsNpArr])
   ```

   Note Feel free to add functions to the *Model* class as needed to structure your model's initialization and execution code. But make sure to keep the pre- and post-processing of the input data and prediction results (done by the *ImageProcessor*) as they are. In the next step you will implement the *ImageProcessor*.
8. Open *contrib_src/processing.py* to implement the *ImageProcessor* class. The *ImageProcessor* inherits from *ImageProcessorBase*, which already has most of the required data I/O processing implemented. Just your model specific pre- and post-processing has to be implemented, to make the *ImageProcessor* work. There are two pre-processing functions and one post-processing function to be filled in. We'll go through each of these functions individually:

1. **_preprocessBeforeConversionToNumpy(self, image)**

   The *ImageProcessorBase* takes care of loading the input image and then calls this function to let you perform pre-processing on the image. The image comming into this function is either a PIL or a SimpleITK object. So *_preprocessBeforeConversionToNumpy* gives you the option to perform pre-processing using PIL or SimpleITK, which might be more convenient than performing pre-processing on the image in numpy format (see next step). If you decide to implement pre-porcessing here, you should implement it for both, PIL and SimpleITK objects. Make sure this function returns the same type of object as it received (PIL in => PIL out, SimpleITK in => SimpleITK out).

   You do not have to implement this. You can delete this function and implement all your pre-processing using the image converted to numpy (see next step).

2. **_preprocessAfterConversionToNumpy(self, npArr)**

   After the image has passed through the previous function, it is automatically converted to a numpy array and then passed into this function. Here you must implement all additional pre-processing and numpy re-formating necessary for your model to perform inference on the numpy array. The numpy array returned by this function should have the right input format for your model (the output of this function is exactly what is returned by `self._imageProcessor.loadAndPreprocess(input)` in *contrib_src/inference.py*).

3. **computeOutput(self, inferenceResults)**

   This function receives the direct output of your model's inference. Here you must implement all post-processing required to prepare the output in a format that is supported by Modelhub.

   You can either output a list of dictionaries, where each dictionary has a "label" element, giving the name of a class, and a "probability" element, giving the probability of that class. For example:

```
result = []
for i in range (len(inferenceResults)):
    obj = {'label': 'Class ' + str(i),
           'probability': float(inferenceResults[i])}
    result.append(obj)
```

   For this you have to specifiy the output type "label*list" in your model's _config.json*.

   Or you can output a numpy array. The output type specified in model's *config.json* will help users (and Modelhub) to interpret the meaning result array:

9. Edit *init/init.json* and add the id of your Docker, so when starting your model, Modelhub knows which Docker to use (and download from DockerHub).

   Optionally also list any additional files that are hosted externally (i.e. not in your model's GitHub repository). Specify origin and the destination within your model's folder structure. This is particularly useful for pre-trained model files, since they can easily be larger than the maximum file size allowed in a GitHub repository.

   When starting a model, Modelhub will first download the model's repository, then download any external files, and then start the Docker specified in this init file.

10. Add your licenses for the model (i.e. everything in the repsoitory except the sample data) and the license for the sample data to *contrib_src/license/model* and *contrib_src/license/sample_data* respectively.

    If you want to publish your model via Modelhub, make sure the licenses allow us to use your code, model, and sample data (most of the popular open source licenses should be fine, for proprietary licenses you might need to give Modelhub and its users explicit permission).

11. (optional) Customize example code in *contrib_src/sandbox.ipynb*. This jupyter notebook is supposed to showcase how to use your model and interpret the output from python. The standard example code in this notebook is very basic and generic. Usually it is much more informative to a user of your model if the example code is tailored to your model.

You can access and run the Sandbox notebook by starting your model via `python start.py YOUR_MODEL_FOLDER_NAME -e`. For this, copy *start.py* from the modelhub repository to the parent folder of your model folder.

12. It is good practice to include the Dockerfiles your used to build the Docker for your model so other users can comprehend what the Docker contains. Create a folder *dockerfiles/* in your local model clone (next to *contrib_src/* and *init/*) and copy the files from steps 1.1. and 1.3. into this folder.

1.3.4 3. Test your model

1. Manually check if your model works.
   1. Copy *start.py* from the modelhub repository to the parent folder of your model folder.
   2. Run `python start.py YOUR_MODEL_FOLDER_NAME` and check if the web app for your model looks and works as expected. **TODO:** Add info on how to use the web app, because the command just starts the REST API, which the web frontend is accessing. ***NOTE*** If your code uses CUDA on a GPU, you have to add the `-g` flag to `start.py` to enforce the use of the GPU version of Docker. This is only required for testing, once your model is added to the index, the right mode (GPU or CPU) is automatically queried. Run `python start.py -h` for more info.
   3. Run `python start.py YOUR_MODEL_FOLDER_NAME -e` and check if the jupyter notebook *contrib_src/sandbox.ipynb* works as expected.
2. Run automatic integration test. This test will perform a few sanity checks to verify that all the basics seem to be working properly. However, passing this test does not mean your model performs correctly (hence the manual checks).
   1. Copy *test_integration.py* from the modelhub repository to the parent folder of your model folder.
   2. Run `python test_integration.py YOUR_MODEL_FOLDER_NAME`. If all tests pass you are good to publish.

      On some platforms and Docker daemon versions communication to the model's Docker container might fail if the Docker is started implicitly by the integration test. If you get obscure errors during test, try starting your model idependently in a different terminal via `python start.py YOUR_MODEL_FOLDER_NAME` and running the test with the "-m" option: `python test_integration.py YOUR_MODEL_FOLDER_NAME -m`.

      If your model needs particularly long to start up, you need to tell the integration test how long to wait before attempting to communicate with the model. Use the "-t" option.

      Check out the documentation of the integration test by calling `python test_integration.py -h`

1.3.5 4. Publish

1. `git clone https://github.com/modelhub-ai/modelhub.git` (or update if you cloned already).
2. Add your model to the model index list *models.json*. If your model needs a GPU to run, add `"gpu" : true` to the parameters for your model. This tells the start script to run the model with GPU acceleration.
3. Send us a pull request.

1.4 Modelhub IO Configuration

1.4.1 Input Configuration for Single Inputs

As a User

If the model only requires a single image or other type of file for inference, you can simply pass a URL or a path to a local file to the API. For example, you can detect objects using YOLO-v3 by running `python start.py yolo-v3` and then use the API like this:

```
http://localhost:80/api/predict?fileurl=http://example.org/cutedogsandcats.jpg
```

The API then returns the prediction in the specified format. For a thorough description of the API, have a look at its documentation.

As a Collaborator submitting a new Model

For single inputs, please create a configuration for your model according to the example configuration. It is important that you keep the key `"single"` in the config, as the API uses this for accessing the dimension constraints when loading an image. Populate the rest of the configuration file as stated in the contribution guide and the schema. Validate your config file against our config schema with a JSON validator, e.g. this one. Take care to choose the right MIME type for your input, this format will be checked by the API when users call the predict function and load a file. We support a few extra MIME types in addition to the standard MIME types:

If you need other types not supported in the standard MIME types and by our extension, please open an issue on Github.

1.4.2 Input Configuration for Multiple Inputs

As a User

When you use a model that needs more than a single input file for a prediction, you have to pass a JSON file with all the inputs needed for that model. You can have a look at an example here. The important points to keep in mind are:

- There has to be a `format` key with `"application/json"` so that the API can handle the file
- Each of the other keys describes one input and has to have a `format` (see the MIME types above) and a `fileurl`

The `fileurl` can contain a path to a local file (which has to be accessible by the Docker container running the model) or can contain a URL to a file on the web. The REST API can handle both and a mixture of local and web links while the Python API can only access local paths. Passing an input file to the REST API would then look like this:

```
http://localhost:80/api/predict?fileurl=http://example.org/fourimagesofdogs.json
```

For a thorough description of the API, have a look at its documentation.

As a Collaborator submitting a new Model

For multiple inputs, please create a configuration for your model according to the example configuration. The `format` key has to be present at the `input` level and must be equal to `application/json` as all input files will be passed in a json to the API.

The other keys stand for one input file each and must contain a valid format (e.g. `application/dicom`) and dimensions. You can additionally add a description for the input.

Populate the rest of the configuration file as stated in the contribution guide and the schema. Validate your config file against our config schema with a JSON validator, e.g. this one. To access the files passed to your model in the `infer` function, use the keys you specified in the configuration and in the input json file. For example, suppose you have an input with key `t1`: You can access the path the the file in `infer` by using the passed dictionary: `input["t1"]["fileurl"]`. This way you can always be sure that you are accessing the right file. ***HINT*** You can implement additional classes for the loading of your images by adding your own class that extends the `ImageLoader` class and add it to the chain of responsibility for loading the images. One good example is the lfb-rwth-brats model.

Additionally, mismatches between the config file and the input file the user passes to the API are automatically checked before the input is passed to your model.

HINT Check out existing models with multiple inputs to see how they implemented the input handling of multiple inputs, for example one of the BraTS models, e.g. lfb-rwth-brats.

1.5 Modelhub APIs

Documentation of the Modelhub REST API and Python API

1.5.1 REST API

The REST API is the main interface to a model packaged with the Modelhub framework. The REST API of a running model can be reached under http://<ip of model>:<port>/api/<call>. For example `http://localhost:80/api/get_config` to retrieve a JSON string with the model configuration.

The REST API is automatically instantiated when you start a model via `python start.py <your model name>`. See the following documentation of the *`ModelHubRESTAPI`* class for a documentation of all available functions.

REST API Class

class `modelhubapi.restapi.`**`ModelHubRESTAPI`**(*model*, *contrib_src_dir*)

`get_config`()

GET method

Returns Model configuration dictionary.

Return type application/json

`get_legal`()

GET method

Returns

All of modelhub's, the model's, and the sample data's legal documents as dictionary. If one (or more) of the legal files don't exist, the error will be logged with the corresponding key. Dictionary keys are:

- modelhub_license
- modelhub_acknowledgements
- model_license

- sample_data_license

Return type application/json

get_model_io()
GET method

Returns The model's input/output sizes and types as dictionary. Convenience function, as this is a subset of what *get_config()* returns

Return type application/json

get_model_files()
GET method

Returns The trained deep learning model in its native format and all its asscociated files in a single zip folder.

Return type application/zip

get_samples()
GET method

Returns List of URLs to all sample files associated with the model.

Return type application/json

predict()
GET/POST method

Returns Prediction result on input data. Return type/format as specified in the model configuration (see *get_model_io()*), and wrapped in json. In case of an error, returns a dictionary with error info.

Return type application/json

GET method

Parameters **fileurl** – URL to input data for prediciton. Input type must match specification in the model configuration (see *get_model_io()*) URL must not contain any arguments and should end with the file extension.

GET Example: :code: *curl -X GET http://localhost:80/api/predict?fileurl=<URL_OF_FILE>*

POST method

Parameters **file** – Input file with data for prediction. Input type must match specification in the model configuration (see *get_model_io()*)

POST Example: :code: *curl -i -X POST -F file=@<PATH_TO_FILE> 'http://localhost:80/api/predict*

predict_sample()
GET method

Performs prediction on sample data.

Note: Currently you cannot use *predict()* for inference on sample data hosted under the same IP as the model API. This function is a temporary workaround. To be removed in the future.

Returns Prediciton result on input data. Return type as specified in the model configuration (see *get_model_io()*), and wrapped in json. In case of an error, returns a dictionary with error info.

Return type application/json

Parameters **`filename`** – File name of the sample data. No folders or URLs.

1.5.2 Python API

The Python API is a convenience interface to a model when you have direct access to the modelhub runtime environment, i.e. when you are inside the Docker running the model. This is, for example, the case if you work with the sandbox Jupyter notebook provided with the model you are running.

When you are working inside the Docker running a model, you can import the Modelhub Python API via `from modelapi import model`. This is a convenience import, which implicitly takes care of initializing the *`ModelHubAPI`* with the model in the current Docker. You would then call the API (e.g. to get the model config) like this `configuration = model.get_config()`.

Python API Class

class `modelhubapi.pythonapi.`**`ModelHubAPI`**(*model*, *contrib_src_dir*)
Generic interface to access a model.

`get_config`()

Returns Model configuration.

Return type dict

`get_legal`()

Returns

All of modelhub's, the model's, and the sample data's legal documents as dictionary. If one (or more) of the legal files don't exist, the error will be logged with the corresponding key. Dictionary keys are:

- modelhub_license
- modelhub_acknowledgements
- model_license
- sample_data_license

Return type dict

`get_model_io`()

Returns The model's input/output sizes and types as dictionary. Convenience function, as this is a subset of what *`get_config()`* returns

Return type dict

`get_samples`()

Returns Folder and file names of sample data bundled with this model. The diconary key "folder" holds the absolute path to the sample data folder in the model container. The key "files" contains a list of all file names in that folder. Join these together to get the full path to the sample files.

Return type dict

`predict`(*input_file_path*, *numpyToFile=True*, *url_root="*)
Preforms the model's inference on the given input.

Parameters

- **`input_file_path`** (*`str or dict`*) – Path to input file to run inference on. Either a direct input file or a json containing paths to all input files needed for the model to predict. The appropriate structure for the json can be found in the documentation. If used directly, you can also pass a dict with the keys.
- **`numpyToFile`** (*`bool`*) – Only effective if prediction is a numpy array. Indicates if numpy outputs should be saved and a path to it is returned. If false, a json-serializable list representation of the numpy array is returned instead. List representations is very slow with large numpy arrays.
- **`url_root`** (*`str`*) – Url root added by the rest api.

Returns Prediction result on input data. Return type/foramt as specified in the model configuration (see *`get_model_io()`*). In case of an error, returns a dictionary with error info.

Return type dict, list, or numpy array

1.6 Modelhub Library

Overview of the classes of the Modelhub library.

1.6.1 Model

class `modelhublib.model.`**`ModelBase`**

Abstract base class for contributer models. Currently this is merely an interface definition that all contributer implemented models have to follow.

`infer`(*input*)

Abstract method. Overwrite this method to implement the inference of a model.

Parameters **`input`** (*`str`*) – Input file name.

Returns Converted inference results into format as defined in the model configuration. Usually should return the result of *`<YourImageProcessor>.computeOutput`*

1.6.2 Pre- and Postprocessing

class `modelhublib.processor.`**`ImageProcessorBase`**(*config*)

Abstract base class for image pre- and postprocessing, thus handeling all data processing before and after the inference.

Several methods of this class have to be implemented in a contributed model. Follow the "Contribute Your Model to Modelhub" guide for detailed instructions.

An image processor handles:

1. Loading of the input image(s).
2. Converting the loaded images to a numpy array
3. Preprocessing the image data (either on the image object or on the numpy array) After this step the data should be prepared to be directly feed to the inference step.
4. Processing the inference result and convert it to the expected output format.

This class already provides loading and conversion of images using PIL and SimpleITK. If you need to support image formats which are not covered by those two, you should implement an additional `ImageLoader` and `ImageConverter`. If you do so, you will also need to overwrite the constructor (__init__) to instantiate your loader and converter and include them in the chain of responsibility. Best practice would be to call the original constructor from your derived class and then change what you need to change.

Parameters **`config`** (`dict`) – Model configuration (loaded from model's config.json)

`loadAndPreprocess`(*input*, *id=None*)

Loads input, preprocesses it and returns a numpy array appropriate to feed into the inference model (4 dimensions: [batchsize, z/color, height, width]).

There should be no need to overwrite this method in a derived class! Rather overwrite the individual preprocessing steps used by this method!

Parameters

- **`input`** (`str`) – Name of the input file to be loaded
- **`id`** (`str or None`) – ID of the input when handling multiple inputs

Returns numpy array appropriate to feed into the inference model (4 dimensions: [batchsize, z/color, height, width])

`computeOutput`(*inferenceResults*)

Abstract method. Overwrite this method to define how to postprocess the inference results computed by the model into a proper output as defined in the model configuration file.

Parameters **`inferenceResults`** – Results of the inference as computed by the model.

Returns Converted inference results into format as defined in the model configuration.

`_load`(*input*, *id=None*)

Performs the actual loading of the image.

There should be no need to overwrite this method in a derived class! Rather implement an additional `ImageLoader` to support further image formats. See also documentation of `ImageProcessorBase` above.

Parameters

- **`input`** (`str`) – Name of the input file to be loaded
- **`id`** (`str or None`) – ID of the input when handling multiple inputs

Returns Image object which type will be the native image object type of the library/handler used for loading (default implementation uses PIL or SimpleITK). Hence it might not always be the same.

`_preprocessBeforeConversionToNumpy`(*image*)

Perform preprocessing on the loaded image object (see `_load()`).

Overwrite this to implement image preprocessing using the loaded image object. If not overwritten, just returns the image object unchanged.

When overwriting this, make sure to handle the possible types appropriately and throw an IOException if you cannot preprocess a certain type.

Parameters **`image`** (type = return of `_load()`) – Loaded image object

Returns Image object which must be of the same type as input image object.

`_convertToNumpy`(*image*)

Converts the image object into a corresponding numpy array with 4 dimensions: [batchsize, z/color, height, width].

There should be no need to overwrite this method in a derived class! Rather implement an additional `ImageConverter` to support further image format conversions. See also documentation of `ImageProcessorBase` above.

Parameters **`image`** – (type = return of `_preprocessBeforeConversionToNumpy()`): Loaded and preproceesed image object.

Returns Representation of the input image as numpy array with 4 dimensions [batchsize, z/color, height, width].

`_preprocessAfterConversionToNumpy`(*npArr*)

Perform preprocessing on the numpy array (the result of _convertToNumpy()).

Overwrite this to implement preprocessing on the converted numpy array. If not overwritten, just returns the input array unchanged.

Parameters **`npArr`** (*`numpy array`*) – input data after conversion by `_convertToNumpy()`

Returns Preprocessed numpy array with 4 dimensions [batchsize, z/color, height, width].

1.6.3 Image Loading

`class` `modelhublib.imageloaders.imageLoader.`**`ImageLoader`**(*config*, *successor=None*)

Abstract base class for image loaders, following chain of responsibility design pattern. For each image loader you should implement a corresponding image converter using `ImageConverter` as base class.

Parameters **`sucessor`** (`ImageLoader`) – Next loader in chain to attempt loading the image if this one fails.

`setSuccessor`(*successor*)

Setting the next loader in chain of responsibility.

Parameters **`sucessor`** (`ImageLoader`) – Next loader in chain to attempt loading the image if this one fails.

`load`(*input*, *id=None*)

Tries to load input and on fail forwards load request to next handler until success or final fail.

There should be no need to overwrite this. Overwrite only `_load()` to load the image type you want to support and let this function as it is to handle the chain of responsibility and errors.

Parameters **`input`** (*`str`*) – Name of the input file to be loaded.

Returns Image object as loaded by `_load()` or a successor load handler.

Raises IOError if input could not be loaded by any load handler in the chain.

`_load`(*input*)

Abstract method. Overwrite to implement loading of the input format you want to support.

When overwriting this, make sure to raise IOError if input cannot be loaded.

Parameters **`input`** (*`str`*) – Name of the input file to be loaded.

Returns Should return image object in the native format of the library using to load it.

`_checkConfigCompliance`(*image*, *id=None*)

Checks if image complies with configuration.

There should be no need to overwrite this. Overwrite only `_getImageDimensions()` to supply the image dims to check against config.

Parameters **`image`** – Image object as loaded by `_load()`

Raises IOError if image dimensions do not comply with configuration.

`_getImageDimensions`(*image*)
Abstract method. Should return the dimensions of the loaded image, should be a 3 tuple (z, y, x).

Overwrite this in an implementation of this interface. This function is used by *`_checkConfigCompliance()`*.

Parameters **`image`** – Image object as loaded by *`_load()`*

Returns Should return image dimensions of the image object.

`class` `modelhublib.imageloaders.pilImageLoader.`**`PilImageLoader`**(*config*, *successor=None*)
Bases: *`modelhublib.imageloaders.imageLoader.ImageLoader`*

Loads common 2d image formats (png, jpg, . . .) using Pillow (PIL).

`_load`(*input*)
Loads input using PIL.

Parameters **`input`** (*`str`*) – Name of the input file to be loaded

Returns PIL.Image object

`_getImageDimensions`(*image*)

Parameters **`image`** (*`PIL.Image`*) – Image as loaded by *`_load()`*

Returns Image dimensions from PIL image object

`class` `modelhublib.imageloaders.sitkImageLoader.`**`SitkImageLoader`**(*config*, *successor=None*)
Bases: *`modelhublib.imageloaders.imageLoader.ImageLoader`*

Loads image formats supported by SimpleITK

`_load`(*input*)
Loads input using SimpleITK.

Parameters **`input`** (*`str`*) – Name of the input file to be loaded

Returns SimpleITK.Image object

`_getImageDimensions`(*image*)

Parameters **`image`** (*`SimpleITK.Image`*) – Image as loaded by *`_load()`*

Returns Image dimensions from SimpleITK image object

1.6.4 Image Conversion

`class` `modelhublib.imageconverters.imageConverter.`**`ImageConverter`**(*successor=None*)
Abstract base class for image converters, following chain of responsibility design pattern. For each image loader derived from `ImageLoader` you should implement a corresponding image converter using this as base class.

Parameters **`sucessor`** (`ImageConverter`) – Next converter in chain to attempt loading the image if this one fails.

`setSuccessor`(*successor*)
Setting the next converter in chain of responsibility.

Parameters **`sucessor`** (`ImageConverter`) – Next converter in chain to attempt loading the image if this one fails.

convert(*image*)

Tries to convert image to numpy and on fail forwards convert request to next handler until sucess or final fail.

There should be no need to overwrite this. Overwrite only `_convert()` to convert the image type you want to support and let this function as it is to handle the chain of responsibility and errors.

Parameters **image** – Image object to convert.

Returns Numpy array as converted by `_convert()` or a successor converter.

Raises IOError if image could not be converted by any converter in the chain.

_convert(*image*)

Abstract method. Overwrite to implement image conversion to numpy array from the image object type you want to support.

When overwriting this, make sure to raise IOError if image cannot be converted.

Parameters **image** – Image object to convert.

Returns Should return image object converted to numpy array with 4 dimensions [batchsize, z/color, height, width]

class modelhublib.imageconverters.pilToNumpyConverter.**PilToNumpyConverter**(*successor=None*)

Bases: `modelhublib.imageconverters.imageConverter.ImageConverter`

Converts PIL.Image objects to Numpy

_convert(*image*)

Parameters **image** (*PIL.Image*) – Image object to convert.

Returns Input image object converted to numpy array with 4 dimensions [batchsize, z/color, height, width]

Raises IOError if input is not of type PIL.Image or cannot be converted for other reasons.

class modelhublib.imageconverters.sitkToNumpyConverter.**SitkToNumpyConverter**(*successor=None*)

Bases: `modelhublib.imageconverters.imageConverter.ImageConverter`

Converts SimpltITK.Image objects to Numpy

_convert(*image*)

Parameters **image** (*SimpleITK.Image*) – Image object to convert.

Returns Input image object converted to numpy array with 4 dimensions [batchsize, z/color, height, width]

Raises IOError if input is not of type SimpleITK.Image or cannot be converted for other reasons.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index