TomOpt

TomOpt Authors

Feb 26, 2024

CONTENTS

This repo provides a library for the differential optimisation of scattering muon tomography systems. For an overview, please read our first publication here.

As a disclaimer, this is a library designed to be extended by users for their specific tasks: e.g. passive volume definition, inference methods, and loss functions. Additionally, optimisation in TomOpt can be unstable, and requires careful tuning by users. This is to say that it is not a polished product for the general public, but rather fellow researchers in the field of optimisation and muon tomography.

If you are interested in using this library seriously, please contact us; we would love to here if you have a specific use-case you wish to work on.

CHAPTER

ONE

OVERVIEW

The TomOpt library is designed to optimise the design of a muon tomography system. The detector system is defined by a set of parameters, which are used to define the geometry of the detectors. The optimisation is performed by minimising a loss function, which is defined by the user. The loss function is evaluated by simulating the muon scattering process through the detector system and passive volumes. The information recorded by the detectors is then passed through an inference system to arrive at a set of task-specific parameters. These are then compared to the ground truth, and the loss is calculated. The gradient of the loss with respect to the detector parameters is then used to update the detector parameters.

The TomOpt library is designed to be modular, and to allow for the easy addition of new inference systems, loss functions, and passive volume definitions. The library is also designed to be easily extensible to new optimisation algorithms, and to allow for the easy addition of new constraints on the detector parameters.

TomOpt consists of several submodules:

- benchmarks: and ongoing collection of concrete implementations and task-specific extensions that are used to test the library on real-world problems.
- inference: provides classes that infer muon-trajectories from detector data, and infer properties of passive volumes from muon-trajectories.
- muon: provides classes for handling muon batches, and generating muons from literature flux-distributions
- optimisation: provides classes for handling the optimisation of detector parameters, and an extensive callback system to modify the optimisation process.
- plotting: various plotting utilities for visualising the detector system, the optimisation process, and results
- volume: contains classes for defining passive volumes and detector systems
- core: core objects used by all parts of the code
- utils: various utilities used throughout the codebase

CHAPTER

TWO

PACKAGE OVERVIEW

2.1 Installation

2.1.1 As a dependency

For dependency usage, `tomopt` can be installed via e.g.

```
pip install tomopt
```

2.1.2 For development

Check out the repo locally:

```
git clone git@github.com:GilesStrong/tomopt.git
cd tomopt
```

For development usage, we use ``*poetry*`` <https://python-poetry.org/docs/#installing-with-the-official-installer>`_ to handle dependency installation. Poetry can be installed via, e.g.

```
curl -sSL https://install.python-poetry.org | python3 -
poetry self update
```

and ensuring that `poetry` is available in your `$PATH`

TomOpt requires `python >= 3.10`. This can be installed via e.g. ``*pyenv*`` <https://github.com/pyenv/pyenv>`_:

```
curl https://pyenv.run | bash
pyenv update
pyenv install 3.10
pyenv local 3.10
```

Install the dependencies:

```
poetry install
poetry self add poetry-plugin-export
poetry config warnings.export false
poetry run pre-commit install
```

Finally, make sure everything is working as expected by running the tests:

```
poetry run pytest tests
```

For those unfamiliar with `poetry`, basically just prepend commands with `poetry run` to use the stuff installed within the local environment, e.g. `poetry run jupyter notebook` to start a jupyter notebook server.. This local environment is basically a python virtual environment. To correctly set up the interpreter in your IDE, use `poetry run which python` to see the path to the correct python executable.

2.2 Examples

A few examples are included to introduce users and developers to the TomOpt library. These take the form of Jupyter notebooks. In `examples/getting_started` there are four ordered notebooks:

- `00_Hello_World.ipynb` aims to show the user the high-level classes in TomOpt and the general workflow.
- `01_Indepth_tutorial_single_cycle.ipynb` aims to show developers what is going on in a single update iteration.
- `02_Indepth_tutotial_optimisation_and_callbacks.ipynb` aims to show users and developers the workings of the callback system in TomOpt
- `03_fixed_budget_mode.ipynb` aims to show users and developers how to optimise such that the detector maintains a constant cost.

In `examples/benchmarks` there is a single notebook that covers the optimisation performed in our first publication, in which we optimised a detector to estimate the fill-height of a ladle furnace at a steel plant. As a disclaimer, this notebook may not fully reproduce our result, and is designed to be used in an interactive manner by experienced users.

2.2.1 Running notebooks in a remote cluster

If you want to run notebooks on a remote cluster but access them on the browser of your local machine, you need to forward the notebook server from the cluster to your local machine.

On the cluster, run:

```
poetry run jupyter notebook --no-browser --port=8889
```

On your local computer, you need to set up a forwarding that picks the flux of data from the cluster via a local port, and makes it available on another port as if the server was in the local machine:

```
ssh -N -f -L localhost:8888:localhost:8889 username@cluster_hostname
```

The layperson version of this command is: *take the flux of info from the port `8889` of `cluster_hostname`, logging in as `username`, get it inside the local machine via the port `8889`, and make it available on the port `8888` as if the jupyter notebook server was running locally on the port `8888`*

You can now point your browser to http://localhost:8888/tree (you will be asked to copy the server authentication token, which is the number that is shown by jupyter when you run the notebook on the server)

If there is an intermediate machine (e.g. a gateway) between the cluster and your local machine, you need to set up a similar port forwarding on the gateway machine. The crucial point is that the input port of each machine must be the output port of the machine before it in the chain. For instance:

```
jupyter notebook --no-browser --port=8889 # on the cluster
ssh -N -f -L localhost:8888:localhost:8889 username@cluster_hostname # on the gateway.␣
↪Makes the notebook running on the cluster port 8889 available on the local port 8888
```

(continues on next page)

(continued from previous page)

```
ssh -N -f -L localhost:8890:localhost:8888 username@gateway_hostname # on your local␣
↪machine. Picks up the server available on 8888 of the gateway and makes it available␣
↪on the local port 8890 (or any other number, e.g. 8888)
```

2.3 External repos

N.B. Most are not currently public

- tomo_deepinfer (contact @GilesStrong for access) separately handles training and model definition of GNNs used for passive volume inference. Models are exported as JIT-traced scripts, and loaded here using the `DeepVolumeInferer` class. We still need to find a good way to host the trained models for easy download.
- mode_muon_tomography_scattering (contact @GilesStrong for access) separately handles conversion of PGeant model from root to HDF5, and Geant validation data from csv to HDF5.
- tomopt_sphinx_theme public. Controls the appearance of the docs.

2.4 Authors

The TomOpt project, and its continued development and support, is the result of the combined work of many people, whose contributions are summarised in the author list

CHAPTER

THREE

PACKAGE DOCUMENTATION

3.1 tomopt package

3.1.1 Subpackages

tomopt.benchmarks package

Subpackages

tomopt.benchmarks.ladle_furnace package

Submodules

tomopt.benchmarks.ladle_furnace.data module

class tomopt.benchmarks.ladle_furnace.data.**LadleFurnacePassiveGenerator**(*volume*, *x0_furnace=0.01782*, *fill_material='hot liquid steel'*, *slag_material='slag'*)

Bases: *AbsPassiveGenerator*

Research tested only: no unit tests

tomopt.benchmarks.ladle_furnace.inference module

class tomopt.benchmarks.ladle_furnace.inference.**EdgeDetLadleFurnaceFillLevelInferrer**(*partial_x0_inferrer*, *volume*, *pipeline=['remove_ladle', 'avg_3d', 'avg_layers', 'avg_1d', 'ridge_1d_0', 'negative', 'max_div_min']*, *add_batch_dim=True*, *output_probs=True*)

Bases: *AbsIntClassifierFromX0*

Research tested only: no unit tests

static avg_1d(*x*)

Return type
Tensor

static avg_3d(*x*)

Return type
Tensor

static avg_layers(*x*)

Return type
Tensor

static edge_det(*x*, *kernel*)

Return type
Tensor

static gauss_1d(*x*)

Return type
Tensor

static gauss_3d(*x*)

Return type
Tensor

laplacian_1d(*x*)

Return type
Tensor

static max_div_min(*x*)

Return type
Tensor

static max_sub_min(*x*)

Return type
Tensor

static negative(*x*)

Return type
Tensor

prewit_1d(*x*)

Return type
Tensor

static remove_ladle(*x*)

Assumes ladle is 1 voxel thick

Return type
Tensor

ridge_1d_0(*x*)

Return type
Tensor

ridge_1d_2(*x*)

Return type
Tensor

ridge_1d_4(*x*)

Return type
Tensor

ridge_1d_8(*x*)

Return type
Tensor

x02probs(*vox_preds*)

Inheriting classes must override this method to convert voxelwise X0 predictions to class probabilities

Parameters
vox_preds (Tensor) – (z,x,y) tensor of voxelwise X0 predictions

Return type
Tensor

Returns
(*) tensor of class probabilities

class tomopt.benchmarks.ladle_furnace.inference.**LinearCorrectionCallback**(*partial_opt*, *init_weight=1.0*, *init_bias=0.0*)

Bases: *Callback*

Research tested only: no unit tests

on_backwards_end()

Runs when the loss for a batch of passive volumes has been backpropagated, but parameters have not yet been updated.

Return type
None

on_train_begin()

Runs when detector fitting begins.

Return type
None

on_volume_batch_begin()

Runs when a new batch of passive volume layouts is begins.

Return type
None

on_x0_pred_end()

Runs after the volume inferrer has made its final prediction, but before the loss is computed.

Return type
None

class tomopt.benchmarks.ladle_furnace.inference.**PocaZLadleFurnaceFillLevelInferrer**(*volume*, *smooth=0.1*)

Bases: *AbsVolumeInferrer*

Research tested only: no unit tests

Computes fill heigh based on weighted average of z of POCAs

compute_efficiency(*scatters*)

Computes the per-muon efficiency, given the individual muon hit efficiencies, as the probability of at least two hits above and below the passive volume.

Parameters
scatters (*ScatterBatch*) – scatter batch containing muons whose efficiency should be computed

Return type
Tensor

Returns
(muons) tensor of muon efficiencies

get_prediction()

Computes the predicted fill level via a weighted average of POCA locations.

Returns
fill-height prediction [m]

Return type
pred

property muon_efficiency: Tensor

Returns: (muons,1) tensor of the efficiencies of the muons

property muon_poca_xyz: Tensor

Returns: (muons,xyz) tensor of PoCA locations

property muon_poca_xyz_unc: Tensor

Returns: (muons,xyz) tensor of PoCA location uncertainties

property n_mu: int

Returns: Total number muons included in the inference

property pred_height: Tensor

Returns: (h) tensor of fill-height prediction

property smooth: Tensor

tomopt.benchmarks.ladle_furnace.loss module

class tomopt.benchmarks.ladle_furnace.loss.**LadleFurnaceIntClassLoss**(*, *pred_int_start=0*, *use_mse*, *target_budget*, *budget_smoothing=10*, *cost_coef=None*, *steep_budget=True*, *debug=False*)

Bases: *VolumeIntClassLoss*

Research tested only: no unit tests

class tomopt.benchmarks.ladle_furnace.loss.**SpreadRangeLoss**

Bases: *Callback*

Research tested only: no unit tests

on_volume_batch_begin()

Runs when a new batch of passive volume layouts is begins.

Return type
None

on_volume_batch_end()

Runs when a batch of passive volume layouts is ends.

Return type
None

on_x0_pred_end()

Runs after the volume inferrer has made its final prediction, but before the loss is computed.

Return type
None

tomopt.benchmarks.ladle_furnace.plotting module

tomopt.benchmarks.ladle_furnace.plotting.**compare_init_optimised_2**(*df_start*, *df_opt_2*, *NAME*)

Return type
None

tomopt.benchmarks.ladle_furnace.plotting.**compare_init_to_optimised**(*df_start*, *df_opt*, *NAME*)

Return type
None

tomopt.benchmarks.ladle_furnace.plotting.**compare_optimised_to_baselines**(*df_bl_1*, *df_bl_2*, *df_opt_2*, *NAME*)

Return type
None

tomopt.benchmarks.ladle_furnace.plotting.**compare_raw_init_to_bias_corrected_init**(*df_start*, *NAME*)

Return type
None

tomopt.benchmarks.ladle_furnace.volume module

tomopt.benchmarks.ladle_furnace.volume.**get_baseline_detector_1**(***, *res=10000.0*, *eff=0.9*, *span=0.8*, *device=device(type='cpu')*)

Return type
ModuleList

tomopt.benchmarks.ladle_furnace.volume.**get_baseline_detector_2**(***, *res=10000.0*, *eff=0.9*, *span=0.8*, *device=device(type='cpu')*)

Return type
ModuleList

tomopt.benchmarks.ladle_furnace.volume.**get_initial_detector**(***, *res=10000.0*, *eff=0.9*, *span=0.8*, *device=device(type='cpu')*)

Return type
ModuleList

tomopt.benchmarks.small_walls package

Submodules

tomopt.benchmarks.small_walls.data module

class tomopt.benchmarks.small_walls.data.**SmallWallsPassiveGenerator**(*volume*, *x0_soil=0.2624248696430881*, *x0_wall=0.08022522418503258*, *stop_k=10*, *turn_k=5*, *min_length=4*, *min_height=4*)

Bases: *AbsPassiveGenerator*

tomopt.benchmarks.small_walls.volume module

tomopt.benchmarks.small_walls.volume.**get_small_walls_volume**(*size=1*, *passive_lwh=tensor([10., 10., 10.])*, *span=4.0*, *res=10000.0*, *eff=1.0*, *det_height=1.0*, *device=device(type='cpu')*)

Return type
Volume

tomopt.benchmarks.small_walls.volume.**get_small_walls_volume_wrapper**(*size=1*, *passive_lwh=tensor([10., 10., 10.])*, *span=4.0*, *res=10000.0*, *eff=1.0*, *det_height=1.0*, *device=device(type='cpu')*)

Return type
PanelVolumeWrapper

tomopt.benchmarks.u_lorry package

Submodules

tomopt.benchmarks.u_lorry.data module

class tomopt.benchmarks.u_lorry.data.**ULorryPassiveGenerator**(*volume*, *u_volume*, *u_prob=0.5*, *fill_frac=0.8*, *x0_lorry=0.01757*, *bkg_materials=['air', 'iron']*)

Bases: *AbsPassiveGenerator*

Research tested only: no unit tests

tomopt.inference package

Submodules

tomopt.inference.scattering module

class tomopt.inference.scattering.**GenScatterBatch**(*mu*, *volume*)

Bases: *ScatterBatch*

Class for computing scattering information from the true hits via incoming/outgoing trajectory fitting.

> **Warning:** This class is intended for diagnostic purposes only. The tracks and scatter variables carry no gradient w.r.t. detector parameters (except z position).

Linear fits are performed separately to all hits associated with layer groups, as indicated by the *pos* attribute of the layers which recorded hits. Currently, the inference methods expect detectors above the passive layer to have *pos=='above'*, and those below the passive volume to have *pos=='below'*. Trajectory fitting is performed using an analytic likelihood minimisation, but no uncertainties on the hits are considered.

Important: The current separation of hits into above and below groups does not allow for e.g. a third set of detectors, since this split is based on the value of the *n_hits_above* attribute.

One instance of this class should created for each *MuonBatch*. As part of the initialisation, muons will be filtered using `_filter_scatters()` in order to avoid NaN/Inf values. This results in direct, in-place changes to the *MuonBatch*.

Since many variables of the scattering can be inferred, but not all are required for further inference downstream, variables, and their uncertainties, are computed on a lazy basis, with memoisation: the values are only computed on the first request (if at all) and then stored in case of further requests.

The dtheta, dphi, and total scattering variables are computed under the assumption of small angular scatterings. An assumption is necessary here, since there is a loss of information in the when the muons undergo scattering in theta and phi: since theta is [0,pi] a negative scattering in theta will always results in a positive theta, but phi can become phi+pi. When inferring the angular scattering, one cannot precisely tell whether instead a large scattering in phi occurred. The total scattering (*total_scatter*) is the quadrature sum of dtheta and dphi, and all three are computed under both hypotheses. The final values of these are chosen using the hypothesis which minimises the total amount of scattering. This assumption has been tested and found to be good.

Parameters

- **mu** (*MuonBatch*) – muons with hits to infer on
- **volume** (*Volume*) – volume through which the muons travelled

class tomopt.inference.scattering.**ScatterBatch**(*mu*, *volume*)

Bases: `object`

Class for computing scattering information from the hits via incoming/outgoing trajectory fitting.

Linear fits are performed separately to all hits associated with layer groups, as indicated by the *pos* attribute of the layers which recorded hits. Currently, the inference methods expect detectors above the passive layer to have *pos*=='*above*', and those below the passive volume to have *pos*=='*below*'. Trajectory fitting is performed using an analytic likelihood minimisation, which considers uncertainties and efficiencies on the hits in x and y.

Important: The current separation of hits into above and below groups does not allow for e.g. a third set of detectors, since this split is based on the value of the *n_hits_above* attribute.

One instance of this class should created for each *MuonBatch*. As part of the initialisation, muons will be filtered using `_filter_scatters()` in order to avoid NaN/Inf gradients or values. This results in direct, in-place changes to the *MuonBatch*.

Since many variables of the scattering can be inferred, but not all are required for further inference downstream, variables, and their uncertainties, are computed on a lazy basis, with memoisation: the values are only computed on the first request (if at all) and then stored in case of further requests.

The dtheta, dphi, and total scattering variables are computed under the assumption of small angular scatterings. An assumption is necessary here, since there is a loss of information in the when the muons undergo scattering in theta and phi: since theta is [0,pi] a negative scattering in theta will always results in a positive theta, but phi can become phi+pi. When inferring the angular scattering, one cannot precisely tell whether instead a large scattering in phi occurred. The total scattering (*total_scatter*) is the quadrature sum of dtheta and dphi, and all three are computed under both hypotheses. The final values of these are chosen using the hypothesis which minimises the total amount of scattering. This assumption has been tested and found to be good.

Parameters

- **mu** (*MuonBatch*) – muons with hits to infer on

- **volume** (*Volume*) – volume through which the muons travelled

property above_gen_hits: Tensor | None

Returns: (muons,hits,xyz) tensor of true hits in the "above" detectors

property above_hit_effs: Tensor | None

Returns: (muons,hits,effs) tensor of hit efficiencies in the "above" detectors

property above_hit_uncs: Tensor | None

Returns: (muons,hits,xyz) tensor of uncertainties on hits in the "above" detectors

property above_hits: Tensor | None

Returns: (muons,hits,xyz) tensor of recorded hits in the "above" detectors

property below_gen_hits: Tensor | None

Returns: (muons,hits,xyz) tensor of true hits in the "below" detectors

property below_hit_effs: Tensor | None

Returns: (muons,hits,eff) tensor of hit efficiencies in the "below" detectors

property below_hit_uncs: Tensor | None

Returns: (muons,hits,xyz) tensor of uncertainties on hits in the "below" detectors

property below_hits: Tensor | None

Returns: (muons,hits,xyz) tensor of recorded hits in the "below" detectors

property dphi: Tensor

Returns: (muons,1) delta phi between incoming & outgoing muons

property dphi_unc: Tensor

Returns: (muons,1) uncertainty on dphi

property dtheta: Tensor

Returns: (muons,1) delta theta between incoming & outgoing muons

property dtheta_unc: Tensor

Returns: (muons,1) uncertainty on dtheta

property dtheta_xy: Tensor

Returns: (muons,xy) delta theta_xy between incoming & outgoing muons in the zx and zy planes

property dtheta_xy_unc: Tensor

Returns: (muons,xy) uncertainty on dtheta_xy

property dxy: Tensor

Returns: (muons,xy) distances in x & y from PoCA to incoming|outgoing muons

property dxy_unc: Tensor

Returns: (muons,xy) uncertainty on dxy

property gen_hits: Tensor | None

Returns: (muons,hits,xyz) tensor of true hits

static get_muon_trajectory(*hits*, *uncs*, *lw*)

Fits a linear trajectory to a group of hits, whilst considering their uncertainties on their xy positions. No uncertainty is considered for z positions of hits. The fit is performed via an analytical likelihood-maximisation.

Important: Muons with <2 hits have NaN trajectory

Parameters

- **hits** (Tensor) – (muons,hits,xyz) tensor of hit positions
- **uncs** (Tensor) – (muons,hits,(unc x,unc y,0)) tensor of hit uncertainties
- **lw** (Tensor) – length and width of the passive layers of the volume

Returns
(muons,xyz) fitted-vector directions start: (muons,xyz) initial point of fitted-vector

Return type
vec

get_scatter_mask()

Return type
Tensor

Returns
(muons) Boolean tensor where True indicates that the PoCA of the muon is located within the passive volume

property hit_effs: Tensor | None

Returns: (muons,hits,eff) tensor of hit efficiencies

property hit_uncs: Tensor | None

Returns: (muons,hits,xyz) tensor of uncertainties on hits

property hits: Dict[str, Dict[str, Tensor]]

Returns: Dictionary of hits, as returned by *get_hits()*

property n_hits_above: int | None

Returns: Number of hits per muon in the "above" detectors

property n_hits_below: int | None

Returns: Number of hits per muon in the "below" detectors

property phi_in: Tensor

Returns: (muons,1) phi of incoming muons

property phi_in_unc: Tensor

Returns: (muons,1) uncertainty on phi_in

property phi_out: Tensor

Returns: (muons,1) phi of outgoing muons

property phi_out_unc: Tensor

Returns: (muons,1) uncertainty on phi_out

plot_scatter(*idx*, *savename=None*)

Plots representation of hits and fitted trajectories for a single muon.

Parameters

- **idx** (int) – index of muon to plot
- **savename** (Optional[Path]) – optional path to save figure to

Return type
None

property poca_xyz: Tensor

Returns: (muons,xyz) xyz location of PoCA

property poca_xyz_unc: Tensor

Returns: (muons,xyz) uncertainty on poca_xyz

property reco_hits: Tensor | None

Returns: (muons,hits,xyz) tensor of recorded hits

property theta_in: Tensor

Returns: (muons,1) theta of incoming muons

property theta_in_unc: Tensor

Returns: (muons,1) uncertainty on theta_in

property theta_msc: Tensor

Returns: (muons,1) theta_msc; the total amount of angular scattering

property theta_msc_unc: Tensor

Returns: (muons,1) uncertainty on total_scatter

property theta_out: Tensor

Returns: (muons,1) theta of outgoing muons

property theta_out_unc: Tensor

Returns: (muons,1) uncertainty on theta_out

property theta_xy_in: Tensor

Returns: (muons,xy) decomposed theta and phi of incoming muons in the zx and zy planes

property theta_xy_in_unc: Tensor

Returns: (muons,xy) uncertainty on theta_xy_in

property theta_xy_out: Tensor

Returns: (muons,xy) decomposed theta and phi of outgoing muons in the zx and zy planes

property theta_xy_out_unc: Tensor

Returns: (muons,xy) uncertainty on theta_xy_out

property total_scatter: Tensor

Returns: (muons,1) theta_msc; the total amount of angular scattering

property total_scatter_unc: Tensor

Returns: (muons,1) uncertainty on total_scatter

property track_in: Tensor | None

Returns: (muons,xyz) incoming xyz vector

property track_out: Tensor | None

Returns: (muons,xyz) outgoing xyz vector

property track_start_in: Tensor | None

Returns: (muons,xyz) initial point of incoming xyz vector

property track_start_out: Tensor | None

Returns: (muons,xyz) initial point of outgoing xyz vector

property xyz_in: Tensor

Returns: (muons,xyz) inferred xy position of muon at the z-level of the top of the passive volume

property xyz_in_unc: Tensor

Returns: (muons,xyz) uncertainty on xyz_in

property xyz_out: Tensor

Returns: (muons,xyz) inferred xy position of muon at the z-level of the bottom of the passive volume

property xyz_out_unc: Tensor

Returns: (muons,xyz) uncertainty on xyz_out

tomopt.inference.volume module

class tomopt.inference.volume.**AbsIntClassifierFromX0**(*partial_x0_inferrer*, *volume*, *output_probs=True*, *class2float=None*)

Bases: *AbsVolumeInferrer*

Abstract base class for inferring integer targets through multiclass classification from voxelwise X0 predictions. Inheriting classes must provide a way to convert voxelwise X0s into class probabilities of the required dimension. Requires a basic inferrer for providing the voxelwise X0 predictions. Optionally, the predictions can be returns as the raw class predictions, or the most probable class. In case of the latter, this class can be optionally be converted to a float value via a user-provided processing function.

Parameters

- **partial_x0_inferrer** (Type[*AbsX0Inferrer*]) – (partial) class to instatiate to provide the voxelwise X0 predictions
- **volume** (*Volume*) – volume through which the muons will be passed
- **output_probs** (bool) – if True, will return the per-class probabilites, otherwise will return the argmax of the probabilities, over the last dimension
- **class2float** (Optional[Callable[[Tensor, *Volume*], Tensor]]) – optional function to convert class indices to a floating value

add_scatters(*scatters*)

Appends a new set of muon scatter vairables. When *get_prediction()* is called, the prediction will be based on all *ScatterBatch* s added up to that point

Return type
None

compute_efficiency(*scatters*)

Compuates the per-muon efficiency according to the method implemented by the X0 inferrer.

Parameters
scatters (*ScatterBatch*) – scatter batch containing muons whose efficiency should be computed

Return type
Tensor

Returns
(muons) tensor of muon efficiencies

get_prediction()

Computes the predicions for the volume. If class probabilities were requested during initialisation, then these will be returned. Otherwise the most probable class will be returned, and this will be converted to a float value if *class2float* is not None.

Returns
(*) volume prediction

Return type
pred

abstract x02probs(*vox_preds*)

Inheriting classes must override this method to convert voxelwise X0 predictions to class probabilities

Parameters
vox_preds (Tensor) – (z,x,y) tensor of voxelwise X0 predictions

Return type
Tensor

Returns
(*) tensor of class probabilities

class tomopt.inference.volume.**AbsVolumeInferrer**(*volume*)

Bases: object

Abstract base class for volume inference.

Inheriting classes are expected to be fed multiple *ScatterBatch* s, via *add_scatters()*, for a single *Volume* and return a volume prediction based on all of the muon batches when *get_prediction()* is called.

Parameters
volume (*Volume*) – volume through which the muons will be passed

add_scatters(*scatters*)

Appends a new set of muon scatter variables. When *get_prediction()* is called, the prediction will be based on all *ScatterBatch* s added up to that point

Return type
None

abstract compute_efficiency(*scatters*)

Inheriting classes must override this method to provide a computation of the per-muon efficiency, given the individual muon hit efficiencies.

Return type
Tensor

abstract get_prediction()

Inheriting classes must override this method to provide a prediction computed using the added scatter batches. E.g. the sum of muon efficiencies.

Return type
Optional[Tensor]

class tomopt.inference.volume.**AbsX0Inferrer**(*volume*)

Bases: *AbsVolumeInferrer*

Abstract base class for inferring the X0 of every voxel in the passive volume.

The inference is based on the PoCA approach of assigning the entirety of the muon scattering to a single point, and the X0 computation is based on inversion of the PDG scattering model described in https://pdg.lbl.gov/2019/reviews/rpp2018-rev-passage-particles-matter.pdf.

Once all scatter batches have been added, the inference proceeds thusly:

- For each muon i, a probability p_ij, is computed according to the probability that the PoCA was located in voxel j.
- These probabilities are computed by integrating over the voxel the PDF of 3 uncorrelated Gaussians centred on the PoCA, with scales equal the uncertainty on the PoCA position in x,y,z.
- p_ij is multiplied by muon efficiency e_i to compute a muon/voxel weight w_ij.
- Inversion of the PDG model gives: $X_0 = \left(\frac{0.0136}{p^{\text{rms}}}\right)^2 \frac{\delta z}{\cos\left(\bar{\theta}^{\text{rms}}\right)} \frac{2}{\theta_{\text{tot.}}^{\text{rms}}}$
- **In order to account for the muon weights and compute different X0s for the voxels whilst using the whole muon population:**
  - Weighted RMSs are computed for each of the scattering terms in the right-hand side of the equation.
  - In addition to the muon weight w_ij, the variances of the squared values of the scattering variables is used to divide w_ij.
- The result is a set of X0 predictions X0_j.

Important: Inversion of the PDG model does NOT account for the natural log term.

Important: To simplify the computation code, this class relies heavily on lazy computation and memoisation; be careful if calling private methods manually.

Parameters
volume (*Volume*) – volume through which the muons will be passed

`get_prediction()`

Computes the predicted X0 per voxel as a (z,x,y) tensor via PDG scatter-model inversion for the provided scatter batches.

Returns
(z,x,y) voxelwise X0 predictions

Return type
pred

`property muon_efficiency: Tensor`

Returns: (muons,1) tensor of the efficiencies of the muons

`property muon_mom: Tensor`

Returns: (muons,1) tensor of the momenta of the muons

`property muon_mom_unc: Tensor`

Returns: (muons,1) tensor of the uncertainty on the momenta of the muons

`property muon_poca_xyz: Tensor`

Returns: (muons,xyz) tensor of PoCA locations

property muon_poca_xyz_unc: Tensor

Returns: (muons,xyz) tensor of PoCA location uncertainties

property muon_probs_per_voxel_zxy: Tensor

> **Warning:** Integration tested only

TODO: Don't assume that poca_xyz uncertainties are uncorrelated TODO: Improve efficiency: currently CDFs are computed multiple times at the same points; could precompute x,y,z probs once, and combine in triples :returns: (muons,z,x,y) tensor of probabilities that the muons' PoCAs were located in the given voxels.

property muon_theta_in: Tensor

Returns: (muons,1) tensor of the thetas of the incoming muons

property muon_theta_in_unc: Tensor

Returns: (muons,1) tensor of the uncertainty on the theta of the incoming muons

property muon_theta_out: Tensor

Returns: (muons,1) tensor of the thetas of the outgoing muons

property muon_theta_out_unc: Tensor

Returns: (muons,1) tensor of the uncertainty on the theta of the outgoing muons

property muon_total_scatter: Tensor

Returns: (muons,1) tensor of total angular scatterings

property muon_total_scatter_unc: Tensor

Returns: (muons,1) tensor of uncertainties on the total angular scatterings

property n_mu: int

Returns: Total number muons included in the inference

property vox_zxy_x0_pred_uncs: Tensor

> **Warning:** Not recommended for use: long calculation; not unit-tested

Returns

(z,x,y) tensor of uncertainties on voxelwise X0s

property vox_zxy_x0_preds: Tensor

Returns: (z,x,y) tensor of voxelwise X0 predictions

static x0_from_scatters(*deltaz*, *total_scatter*, *theta_in*, *theta_out*, *mom*)

Computes the X0 of a voxel, by inverting the PDG scattering model in terms of the scattering variables

> **Important:** Inversion of the PDG model does NOT account for the natural log term.

Parameters

- **deltaz** (`float`) – height of the voxels

- **total_scatter** (Tensor) – (voxels,1) tensor of the (RMS of the) total angular scattering of the muon(s)
- **theta_in** (Tensor) – (voxels,1) tensor of the (RMS of the) theta of the muon(s), as inferred using the incoming trajectory/ies
- **theta_out** (Tensor) – (voxels,1) tensor of the (RMS of the) theta of the muon(s), as inferred using the outgoing trajectory/ies
- **mom** (Tensor) – (voxels,1) tensor of the (RMS of the) momentum/a of the muon(s)

Return type
Tensor

Returns
(voxels,1) estimated X0 in metres

class tomopt.inference.volume.**DenseBlockClassifierFromX0s**(*n_block_voxels*, *partial_x0_inferrer*, *volume*, *use_avgpool=True*, *cut_coef=10000.0*, *ratio_offset=-1.0*, *ratio_coef=1.0*)

Bases: *AbsVolumeInferrer*

Class for inferreing the presence of a small amount of denser material in the passive volume.

Transforms voxel-wise X0 preds into binary classification statistic under the hypothesis of a small, dense block against a light-weight background. This test statistic, s is computed as:

$$r = 2\frac{\bar{X0}_{0,\text{bkg}} - \bar{X0}_{0,\text{blk}}}{\bar{X0}_{0,\text{bkg}} + \bar{X0}_{0,\text{blk}}} s = \sigma(a(r+b))$$

where $\bar{X0}_{0,\text{blk}}$ is the mean X0 of the N lowest X0 voxels, and $\bar{X0}_{0,\text{bkg}}$ is the mean X0 of the remaining voxels. a and b are rescaling coefficients and offsets.

This results in a differentiable value constrained beween 0 and 1, with values near 0 indicating that no relatively dense material is present, and values nearer 1 indicating that it is present. In case it is expected that the dense material forms a contiguous block, the voxelwise X0s can be blurred via a stride-1 kernel-size-3 average pooling.

In actuality, the "cut" on X0s into background and block is implemented as a sigmoid weight, centred at the necessary kth value of the X0. This means that the test statisitc is also differentiable w.r.t. the cut.

Parameters

- **n_block_voxels** (int) – number of voxels expected to be occupied by the dense material, if present
- **partial_x0_inferrer** (Type[*AbsX0Inferrer*]) – (partial) class to instatiate to provide the voxelwise X0 predictions
- **volume** (*Volume*) – volume through which the muons will be passed
- **use_avgpool** (bool) – wether to blur voxelwise X0 predicitons with a stride-1 kernel-size-3 average pooling useful when the dense material is expected to form a contiguous block
- **cut_coef** (float) – the "sharpness" of the sigmoid weight that splits voxels into block and background. Higher values results in a sharper cut.
- **ratio_offset** (float) – additive constant for the X0 ratio
- **ratio_coef** (float) – multiplicative coefficient for the offset X0 ratio

add_scatters(*scatters*)

Appends a new set of muon scatter vairables. When *get_prediction()* is called, the prediction will be based on all *ScatterBatch* s added up to that point

Return type
None

compute_efficiency(*scatters*)

Compuates the per-muon efficiency according to the method implemented by the X0 inferrer.

Parameters
scatters (*ScatterBatch*) – scatter batch containing muons whose efficiency should be computed

Return type
Tensor

Returns
(muons) tensor of muon efficiencies

get_prediction()

Computes the test statistic for the volume, with values near 0 indicating that no relatively dense material is present, and values nearer 1 indicating that it is present.

Returns
(1,1,1) volume prediction

Return type
pred

class tomopt.inference.volume.**PanelX0Inferrer**(*volume*)

Bases: *AbsX0Inferrer*

Class for inferring the X0 of every voxel in the passive volume using hits recorded by *PanelDetectorLayer* s.

The inference is based on the PoCA approach of assigning the entirety of the muon scattering to a single point, and the X0 computation is based on inversion of the PDG scattering model described in https://pdg.lbl.gov/2019/reviews/rpp2018-rev-passage-particles-matter.pdf.

Once all scatter batches have been added, the inference proceeds thusly:

- For each muon i, a probability p_ij, is computed according to the probability that the PoCA was located in voxel j.
- These probabilities are computed by integrating over the voxel the PDF of 3 uncorrelated Gaussians centred on the PoCA, with scales equal the uncertainty on the PoCA position in x,y,z.
- p_ij is multiplied by muon efficiency e_i to compute a muon/voxel weight w_ij.
- Inversion of the PDG model gives: $X_0 = \left(\frac{0.0136}{p^{\text{rms}}}\right)^2 \frac{\delta z}{\cos\left(\theta^{\text{rms}}\right)} \frac{2}{\theta^{\text{rms}}_{\text{tot.}}}$
- **In order to account for the muon weights and compute different X0s for the voxels whilst using the whole muon population:**
  - Weighted RMSs are computed for each of the scattering terms in the right-hand side of the equation.
  - In addition to the muon weight w_ij, the variances of the squared values of the scattering variables is used to divide w_ij.
- The result is a set of X0 predictions X0_j.

Important: Inversion of the PDG model does NOT account for the natural log term.

Important: To simplify the computation code, this class relies heavily on lazy computation and memoisation; be careful if calling private methods manually.

Parameters
volume (*Volume*) – volume through which the muons will be passed

TODO: refactor this to be provided to volume inference as a callable

compute_efficiency(*scatters*)

Computes the per-muon efficiency, given the individual muon hit efficiencies, as the probability of at least two hits above and below the passive volume.

Parameters
scatters (*ScatterBatch*) – scatter batch containing muons whose efficiency should be computed

Return type
Tensor

Returns
(muons) tensor of muon efficiencies

tomopt.muon package

Submodules

tomopt.muon.generation module

class tomopt.muon.generation.**AbsMuonGenerator**(*x_range*, *y_range*, *fixed_mom=5.0*, *energy_range=(0.5, 500)*, *theta_range=(0, 1.2217304763960306)*)

Bases: object

Abstract generator base class implementing core functionality. Inheriting classes should override the *flux* method.

Once initialised, the object can be called, or it's *generate_set* method called, to generate a set of initial muon kinematics. Each muon will have a starting x and y position sampled uniformly within a defined region. Theta and momentum will be defined by sampling the defined flux model.

Parameters

- **x_range** (Tuple[float, float]) – range in metres of the absolute initial x-position in the volume reference frame over which muons can be generated
- **y_range** (Tuple[float, float]) – range in metres of the absolute initial y-position in the volume reference frame over which muons can be generated
- **fixed_mom** (Optional[float]) – if not None, will only generate muons with the specified momentum in GeV
- **energy_range** (Tuple[float, float]) – if fixed_mom is None, muons will have initial momentum sampled according to the flux model in the specified range in GeV

- **theta_range** (Tuple[float, float]) – muons will have initial theta sampled according to the flux model in the specified range in radians

abstract flux(*energy*, *theta*)

Inheriting classes should override this to implement their flux model for the supplied pairs of energies and thetas

Parameters

- **energy** (Union[float, ndarray]) – energy values at which to compute the flux, in GeV
- **theta** (Union[float, ndarray]) – theta values at which to compute the flux, in radians

Return type
Union[float, ndarray]

Returns
muon flux for every energy & theta pair

classmethod from_volume(*volume*, *min_angle=0.2617993877991494*, *fixed_mom=5.0*, *energy_range=(0.5, 500)*, *theta_range=(0, 1.2217304763960306)*)

Class method to initialise x and y ranges of muon generation from the passive volume. Heuristically computes x,y generation range as (0-d,x+d), (0-d,y+d). Where d is such that a muon generated at (0-d,1) will only hit the last layer of the passive volume if it's initial angle is at least min_angle. This balances a trade-off between generation efficiency and generator realism.

Parameters

- **volume** (*Volume*) – *Volume* through which the muons will pass
- **min_angle** (float) – the minimum theta angle that a muon generated at the extreme x or y boundary would require to hit at least the last passive layer of the passive volume, if it's phi angle were to point directly towards the passive volume.
- **fixed_mom** (Optional[float]) – if not None, will only generate muons with the specified momentum in GeV
- **energy_range** (Tuple[float, float]) – if fixed_mom is None, muons will have initial momentum sampled according to the flux model in the specified range in GeV
- **theta_range** (Tuple[float, float]) – muons will have initial theta sampled according to the flux model in the specified range in radians

Return type
AbsMuonGenerator

generate_set(*n_muons*)

Generates a set of muons as a rank-2 tensor of shape (n_muons, 5), with initial kinematic variables [x, y, momentum, theta, phi]. Theta and, optionally, momentum are sampled from the flux model. x and y are sampled uniformly from the defined ranges. Phi is sampled uniformly from [0,2pi].

Parameters
n_muons (int) – number of muons to generate

Return type
Tensor

Returns
Rank-2 tensor of shape (n_muons, 5), with initial kinematic variables [x, y, momentum, theta, phi]

class tomopt.muon.generation.**MuonGenerator2015**(*x_range*, *y_range*, *fixed_mom=5.0*, *energy_range=(0.5, 500)*, *theta_range=(0, 1.2217304763960306)*)

Bases: *AbsMuonGenerator*

Provides muon generator for sampling initial muon kinematics according to Guan et al. 2015 (arXiv:1509.06176).

Once initialised, the object can be called, or it's *generate_set* method called, to generate a set of initial muon kinematics. Each muon will have a starting x and y position sampled uniformly within a defined region. Theta and momentum will be defined by sampling the defined flux model.

Parameters

- **x_range** (Tuple[float, float]) – range in metres of the absolute initial x-position in the volume reference frame over which muons can be generated
- **y_range** (Tuple[float, float]) – range in metres of the absolute initial y-position in the volume reference frame over which muons can be generated
- **fixed_mom** (Optional[float]) – if not None, will only generate muons with the specified momentum in GeV
- **energy_range** (Tuple[float, float]) – if fixed_mom is None, muons will have initial momentum sampled according to the flux model in the specified range in GeV
- **theta_range** (Tuple[float, float]) – muons will have initial theta sampled according to the flux model in the specified range in radians

P1 = 0.102573

P2 = -0.068287

P3 = 0.958633

P4 = 0.0407253

P5 = 0.817285

flux(*energy*, *theta*)

Function returning modified Gaisser formula for cosmic muon flux given energy (float/np.array) and incidence angle (float/np.array) Uses model defined in Guan et al. 2015 (arXiv:1509.06176)

Parameters

- **energy** (Union[float, ndarray]) – energy values at which to compute the flux, in GeV
- **theta** (Union[float, ndarray]) – theta values at which to compute the flux, in radians

Return type
Union[float, ndarray]

Returns
muon flux for every energy & theta pair

class tomopt.muon.generation.**MuonGenerator2016**(*x_range*, *y_range*, *fixed_mom=5.0*, *energy_range=(0.5, 500)*, *theta_range=(0, 1.2217304763960306)*)

Bases: *AbsMuonGenerator*

Provides muon generator for sampling initial muon kinematics according to Shukla and Sanskrith 2018 arXiv:1606.06907

Once initialised, the object can be called, or it's *generate_set* method called, to generate a set of initial muon kinematics. Each muon will have a starting x and y position sampled uniformly within a defined region. Theta and momentum will be defined by sampling the defined flux model.

Parameters

- **x_range** (Tuple[float, float]) – range in metres of the absolute initial x-position in the volume reference frame over which muons can be generated
- **y_range** (Tuple[float, float]) – range in metres of the absolute initial y-position in the volume reference frame over which muons can be generated
- **fixed_mom** (Optional[float]) – if not None, will only generate muons with the specified momentum in GeV
- **energy_range** (Tuple[float, float]) – if fixed_mom is None, muons will have initial momentum sampled according to the flux model in the specified range in GeV
- **theta_range** (Tuple[float, float]) – muons will have initial theta sampled according to the flux model in the specified range in radians

E_0 = 3.87

E_c = 0.5

I_0 = 88.0

N = 38.1938

Rod = 174.0

epinv = 0.00117096018735363

flux(*energy*, *theta*)

Function returning modified Gaisser formula for cosmic muon flux given energy (float/np.array) and incidence angle (float/np.array) Uses model defined in Shukla and Sanskrith 2018 arXiv:1606.06907

Parameters

- **energy** (Union[float, ndarray]) – energy values at which to compute the flux, in GeV
- **theta** (Union[float, ndarray]) – theta values at which to compute the flux, in radians

Return type
Union[float, ndarray]

Returns
muon flux for every energy & theta pair

n = 3

tomopt.muon.muon_batch module

class tomopt.muon.muon_batch.**MuonBatch**(*xy_p_theta_phi*, *init_z*, *device=device(type='cpu')*)

Bases: object

Container class for a batch of many muons, defined by their position and kinematics.

Each muon has its own:

- x, y, and z position in metres, which are absolute coordinates in the volume frame.

- theta, the angle in radians [0,pi) between the muon trajectory and the negative z-axis in the volume frame muons with a theta > pi/2 (i.e. travel upwards) may be removed automatically
- phi, the anticlockwise angle in radians [0,2pi) between the muon trajectory and the positive x-axis, in the x-y plane of the volume frame.
- momentum (mom), the absolute value of the muon momentum in GeV

Muon properties should not be updated manually. Instead, call:

- *.propagate_dz_dz(dz)* to update the x,y,z positions of the muons for a given propagation dz in the z-axis.
- *.propagate_dz_d(d)* to update the x,y,z positions of the muons for a given propagation d in the muons' trajectories.
- *.scatter_dxy(dx_vol, dy_vol, mask)* to shift the x,y positions of the muons, for which the values of the optional Boolean mask is true, by the specified amount.
- *.scatter_dtheta_dphi(dtheta_vol, dphi_vol, mask)* to alter the theta,phi angles of the muons, for which the values of the optional Boolean mask is true, by the specified amount.

Important: Muon momenta is currently constant

Important: Eventually the muon batch will be extended to store information about the inferred momentum of the muons *reco_mom*. However currently the *reco_mom* property will return the TRUE momentum of the muons, with no simulation of measurement precision.

By default, the *MuonBatch* class only contains the current position of the muons, however the *.snapshot_xyz* method can be used to store the xy positions of the muons at any time, to a dictionary with float z-position keys, *xyz_hist*.

In addition to storing the properties of the muons, the *MuonBatch* class is also used to store the detector hits associated with each muon. Hits may be added via the *.append_hits* method, and stored in the *_hits* attribute. Hits can then be retrieved by the *.get_hits* method.

Parameters

- **xy_p_theta_phi** (`Tensor`) – (N_muon, 5) tensor, with xy [m], p [GeV], theta [r] (0, pi/2) defined w.r.t z axis, phi [r] (0, 2pi) defined anticlockwise from x axis
- **init_z** (`Union[Tensor, float]`) – initial z position of all muons in the batch
- **device** (`device`) – device on which to place the muon tensors

append_hits(*hits*, *pos*)

Record hits to *_hits*.

Parameters

- **hits** (`Dict[str, Tensor]`) – dictionary of 'reco_xy', 'gen_xy', 'z' keys to (muons, *) tensors.
- **pos** (`str`) – Position of detector array in which the hits were recorded, currently either 'above' or 'below'.

Return type

`None`

copy()

Creates a copy of the muon batch at the current position and trajectories. Tensors are detached and cloned.

Important: This does NOT copy of hits

Return type
MuonBatch

Returns
New *MuonBatch* with xyz, and theta,phi equal to those of the current *MuonBatch*.

dtheta(*theta_ref*)

Computes absolute difference in the theta between the muons and the supplied theta angles

Parameters
theta_ref (Tensor) – (N,) tensor to compare with the muon theta values

Return type
Tensor

Returns
Absolute difference between muons' theta and the supplied reference theta

dtheta_x(*theta_ref_x*)

Computes absolute difference in the theta_x between the muons and the supplied theta_x angles

Parameters
theta_ref_x (Tensor) – (N,) tensor to compare with the muon theta_x values

Return type
Tensor

Returns
Absolute difference between muons' theta_x and the supplied reference theta_x

dtheta_y(*theta_ref_y*)

Computes absolute difference in the theta_y between the muons and the supplied theta_y angles

Parameters
theta_ref_y (Tensor) – (N,) tensor to compare with the muon theta_y values

Return type
Tensor

Returns
Absolute difference between muons' theta_y and the supplied reference theta_y

filter_muons(*keep_mask*)

Removes all muons, and their associated hits, except for muons specified as True in *keep_mask*.

Parameters
keep_mask (Tensor) – (N,) Boolean tensor. Muons with False elements will be removed, along with their hits.

Return type
None

`get_hits`(*xy_low=None*, *xy_high=None*)

Retrieve the recorded hits for the muons, optionally only for muons between the specified xy ranges. For ease of use, the list of hits are stacked into single tensors, resulting in a dictionary mapping detector-array position to a dictionary mapping hit variables to (N_muons, N_hits, *) tensors.

Parameters

- **xy_low** (Union[Tuple[float, float], Tensor, None]) – (2,N) optional lower limit on xy positions
- **xy_high** (Union[Tuple[float, float], Tensor, None]) – (2,N) optional upper limit on xy positions

Return type
Dict[str, Dict[str, Tensor]]

Returns
Hits, a dictionary mapping detector-array position to a dictionary mapping hit variables to (N_muons, N_hits, *) tensors.

`get_xy_mask`(*xy_low*, *xy_high*)

Computes a (N,) Boolean tensor, with True values corresponding to muons which are within the supplied ranges in xy.

Parameters

- **xy_low** (Union[Tuple[float, float], Tensor, None]) – (2,N) optional lower limit on xy positions
- **xy_high** (Union[Tuple[float, float], Tensor, None]) – (2,N) optional upper limit on xy positions

Return type
Tensor

Returns
(N,) Boolean mask with True values corresponding to muons which are with xy positions >= xy_low and < xy_high

`property mom: Tensor`

`property muons: Tensor`

`p_dim = 3`

`ph_dim = 5`

`property phi: Tensor`

`static phi_from_theta_xy`(*theta_x*, *theta_y*)

Computes the phi angle from theta_x and theta_y.

Important: This function does NOT work if theta is > pi/2

Parameters

- **theta_x** (Tensor) – angle from the negative z-axis in the xz plane
- **theta_y** (Tensor) – angle from the negative z-axis in the yz plane

Return type
Tensor

Returns
phi, the anti-clockwise angle from the positive x axis, in the xy plane

propagate_d(*d*, *mask=None*)

Propagates all muons in their direction of flight by the specified distances.

Parameters

- **d** (Union[Tensor, float]) – (1,) or (N,) distance(s) in metres to move.
- **mask** (Optional[Tensor]) – (N,) Boolean tensor. If not None, only muons with True mask elements are altered.

Return type
None

propagate_dz(*dz*, *mask=None*)

Propagates all muons in their direction of flight such that afterwards they will all have moved a specified distance in the negative z direction.

Parameters

- **dz** (Union[Tensor, float]) – distance in metres to move in the negative z direction, i.e. a positive dz results in the muons travelling downwards.
- **mask** (Optional[Tensor]) – (N,) Boolean tensor. If not None, only muons with True mask elements are altered.

Return type
None

property reco_mom: Tensor

remove_upwards_muons()

Removes muons, and their hits, if their theta >= pi/2, i.e. they are travelling upwards after a large scattering. Should be run after any changes to theta, but make sure that references (e.g. masks) to the complete set of muons are no longer required.

Return type
None

scatter_dtheta_dphi(*dtheta_vol=None*, *dphi_vol=None*, *mask=None*)

Changes the trajectory of the muons in theta-phi by the specified amounts, with no change in their x,y,z positions. If a mask is supplied, then only muons with True mask elements are altered.

Parameters

- **dtheta_vol** (Optional[Tensor]) – (N,) tensor of angular changes in theta
- **dphi_vol** (Optional[Tensor]) – (N,) tensor of angular changes in phi
- **mask** (Optional[Tensor]) – (N,) Boolean tensor. If not None, only muons with True mask elements are altered.

Return type
None

scatter_dtheta_xy(*dtheta_x_vol=None*, *dtheta_y_vol=None*, *mask=None*)

Changes the trajectory of the muons in theta-phi by the specified amounts in dtheta_xy, with no change in their x,y,z positions. If a mask is supplied, then only muons with True mask elements are altered.

Parameters

- **dtheta_x_vol** (Optional[Tensor]) – (N,) tensor of angular changes in theta_x
- **dtheta_y_vol** (Optional[Tensor]) – (N,) tensor of angular changes in theta_y
- **mask** (Optional[Tensor]) – (N,) Boolean tensor. If not None, only muons with True mask elements are altered.

Return type
None

scatter_dxyz(*dx_vol=None*, *dy_vol=None*, *dz_vol=None*, *mask=None*)

Displaces the muons in xyz by the specified amounts. If a mask is supplied, then only muons with True mask elements are displaced.

Parameters

- **dx_vol** (Optional[Tensor]) – (N,) tensor of displacements in x
- **dy_vol** (Optional[Tensor]) – (N,) tensor of displacements in y
- **dz_vol** (Optional[Tensor]) – (N,) tensor of displacements in z
- **mask** (Optional[Tensor]) – (N,) Boolean tensor. If not None, only muons with True mask elements are displaced.

Return type
None

snapshot_xyz()

Store the current xy positions of the muons in *.xyz_hist*, indexed by the current z position.

Return type
None

th_dim = 4

property theta: Tensor

static theta_from_theta_xy(*theta_x*, *theta_y*)

Computes the theta angle from theta_x and theta_y.

Important: This function does NOT work if theta is > pi/2

Parameters

- **theta_x** (Tensor) – angle from the negative z-axis in the xz plane
- **theta_y** (Tensor) – angle from the negative z-axis in the yz plane

Return type
Tensor

Returns
theta, the anti-clockwise angle from the negative z axis, in the xyz plane

property theta_x: Tensor

static theta_x_from_theta_phi(*theta*, *phi*)

Computes the angle from the negative z-axis in the xz plane from theta and phi

Important: This function does NOT work if theta is > pi/2

Parameters

- **theta** (Tensor) – the anti-clockwise angle from the negative z axis, in the xyz plane
- **phi** (Tensor) – the anti-clockwise angle from the positive x axis, in the xy plane

Return type
Tensor

Returns
theta_x, the angle from the negative z-axis in the xz plane

property theta_xy: Tensor

property theta_y: Tensor

static theta_y_from_theta_phi(*theta*, *phi*)

Computes the angle from the negative z-axis in the yz plane from theta and phi

Important: This function does NOT work if theta is > pi/2

Parameters

- **theta** (Tensor) – the anti-clockwise angle from the negative z axis, in the xyz plane
- **phi** (Tensor) – the anti-clockwise angle from the positive x axis, in the xy plane

Return type
Tensor

Returns
theta_y, the angle from the negative z-axis in the yz plane

property upwards_muons: Tensor

property x: Tensor

x_dim = 0

property xy: Tensor

property xyz: Tensor

property xyz_hist: List[Tensor]

property y: Tensor

y_dim = 1

property z: Tensor

z_dim = 2

tomopt.optimisation package

Subpackages

tomopt.optimisation.callbacks package

Submodules

tomopt.optimisation.callbacks.callback module

class tomopt.optimisation.callbacks.callback.**Callback**

Bases: object

Implements the base class from which all callback should inherit. Callbacks are used as part of the fitting, validation, and prediction methods of *AbsVolumeWrapper*. They can interject at various points, but by default do nothing. Please check in the *AbsVolumeWrapper* to see when exactly their interjections are called.

When writing new callbacks, the VolumeWrapper they are associated with will be their *wrapper* attribute. Their wrapper will have a *fit_params* attribute (*FitParams*) which is a data-class style object. It contains all the objects associated with the fit and predictions, including other callbacks. Callback interjections should read/write to *wrapper.fit_params*, rather than returning values.

Accounting for the interjection calls (*on_*_begin* & *on_*_end*), the full optimisation loop is:

1. Associate callbacks with wrapper (*set_wrapper*)
2. *on_train_begin*
3. **for epoch in *n_epochs*:**
    A. *state* = "train"
    B. *on_epoch_begin*
    C. **for *p*, *passive* in enumerate(*trn_passives*):**
        a. **if *p* % *passive_bs* == 0:**
            i. *on_volume_batch_begin*
            ii. *loss* = 0
        b. load *passive* into passive volume
        c. *on_volume_begin*
        d. **for muon_batch in range(*n_mu_per_volume*//*mu_bs*):**
            i. *on_mu_batch_begin*
            ii. Irradiate volume with *mu_bs* muons
            iii. Infer scatter locations
            iv. *on_scatter_end*
            v. Infer x0 and append to list of x0 predictions

vi. *on_mu_batch_end*

e. *on_x0_pred_begin*

f. Compute overall x0 prediction

g. *on_x0_pred_end*

h. Compute loss based on precision and cost, and add to *loss*

i. **if *p `+1 % `passive_bs* == 0:**

i. *loss* = *loss/passive_bs*

ii. *on_volume_batch_end*

iii. Zero parameter gradients

iv. *on_backwards_begin*

v. Backpropagate *loss* and compute parameter gradients

vi. *on_backwards_end*

vii. Update detector parameters

viii. Ensure detector parameters are within physical boundaries (*AbsDetectorLayer.conform_detector*) viv. *loss* = 0

j. **if len(*trn_passives*)-(*p `+1) < `passive_bs*:**

i. Break

D. *on_epoch_end*

E. *state* = "valid"

F. *on_epoch_begin*

G. **for *p*, *passive* in enumerate(*val_passives*):**

a. **if *p* % *passive_bs* == 0:**

i. *on_volume_batch_begin*

ii. *loss* = 0

b. *on_volume_begin*

c. **for muon_batch in range(*n_mu_per_volume//mu_bs*):**

i. *on_mu_batch_begin*

ii. Irradiate volume with *mu_bs* muons

iii. Infer scatter locations

iv. *on_scatter_end*

v. Infer x0 and append to list of x0 predictions

vi. *on_mu_batch_end*

d. *on_x0_pred_begin*

e. Compute overall x0 prediction

f. *on_x0_pred_end*

g. Compute loss based on precision and cost, and add to *loss*

h. **if** ***p`+1 % `passive_bs*** **== 0:**

i. *loss = loss/passive_bs*

ii. *on_volume_batch_end*

i. **if len(*val_passives*)-(*p`+1*) < *`passive_bs*:**

i. Break

H. *on_epoch_end*

4. *on_train_end*

`on_backwards_begin`()

Runs when the loss for a batch of passive volumes has been computed, but not yet backpropagated.

Return type
None

`on_backwards_end`()

Runs when the loss for a batch of passive volumes has been backpropagated, but parameters have not yet been updated.

Return type
None

`on_epoch_begin`()

Runs when a new training or validations epoch begins.

Return type
None

`on_epoch_end`()

Runs when a training or validations epoch ends.

Return type
None

`on_mu_batch_begin`()

Runs when a new batch of muons begins.

Return type
None

`on_mu_batch_end`()

Runs when a batch muons ends and scatters have been added to the volume inferrer.

Return type
None

`on_pred_begin`()

Runs when the wrapper is about to begin in prediction mode.

Return type
None

`on_pred_end`()

Runs when the wrapper has finished in prediction mode.

Return type
None

on_scatter_end()

Runs when a scatters for the latest muon batch have been computed, but not yet added to the volume inferrer.

Return type
None

on_step_end()

Runs when the parameters have been updated.

Return type
None

on_train_begin()

Runs when detector fitting begins.

Return type
None

on_train_end()

Runs when detector fitting ends.

Return type
None

on_volume_batch_begin()

Runs when a new batch of passive volume layouts is begins.

Return type
None

on_volume_batch_end()

Runs when a batch of passive volume layouts is ends.

Return type
None

on_volume_begin()

Runs when a new passive volume layout is loaded.

Return type
None

on_volume_end()

Runs when a passive volume layout has been predicted.

Return type
None

on_x0_pred_begin()

Runs when the all the muons for a volume have propagated, and the volume inferrer is about to make its final prediction.

Return type
None

on_x0_pred_end()

Runs after the volume inferrer has made its final prediction, but before the loss is computed.

Return type
None

set_wrapper(*wrapper*)

Parameters
wrapper (*AbsVolumeWrapper*) – Volume wrapper to associate with the callback

Return type
None

wrapper: Optional[*AbsVolumeWrapper*] = None

tomopt.optimisation.callbacks.cyclic_callbacks module

class tomopt.optimisation.callbacks.cyclic_callbacks.**CyclicCallback**

Bases: *Callback*

tomopt.optimisation.callbacks.data_callbacks module

class tomopt.optimisation.callbacks.data_callbacks.**MuonResampler**

Bases: *Callback*

Resamples muons to only include those which will impact the passive volume at some point, even if they only hit the bottom layer.

static check_mu_batch(*mu*, *volume*)

Checks the provided muon batch to determine which muons will impact the passive volume at any point

Parameters

- **mu** (*MuonBatch*) – incoming batch of muons
- **volume** (*Volume*) – Volume containing the passive volume to test against

Return type
Tensor

Returns
(muons) Boolean tensor where True indicates that the muon will hit the passive volume

on_mu_batch_begin()

Resamples muons prior to propagation through the volume such that all muons will hit the passive volume.

TODO Add check for realistic validation

Return type
None

static resample(*mus*, *volume*, *gen*)

Resamples muons until all muons will hit the passive volume.

Parameters

- **mus** (Tensor) – xy_p_theta_phi tensor designed to initialise a *MuonBatch*
- **volume** (*Volume*) – Volume containing the passive volume to test against
- **gen** (*AbsMuonGenerator*) – Muon generator for sampling replacement muons

Return type
Tensor

Returns
xy_p_theta_phi tensor designed to initialise a *MuonBatch*

tomopt.optimisation.callbacks.detector_callbacks module

class tomopt.optimisation.callbacks.detector_callbacks.**PanelCentring**

Bases: *Callback*

Callback class for panel centring in the optimisation process.

This callback is used to centre the panels of PanelDetectorLayer objects by setting their xy coordinates to the mean xy value of all panels in the layer.

This update takes place after the panel positions have been updated in the optimisation process.

on_step_end()

Updates the xy coordinates of all panels in the PanelDetectorLayer objects after they have be updated, based on their current mean xy position.

Return type
None

class tomopt.optimisation.callbacks.detector_callbacks.**PanelUpdateLimiter**(*max_xy_step=None*, *max_z_step=None*, *max_xy_span_step=None*)

Bases: *Callback*

Limits the maximum difference that optimisers can make to panel parameters, to prevent them from being affected by large updates from anomolous gradients. This is enacted by a hard-clamping based on the initial and final parameter values before/after each update step.

Parameters

- **max_xy_step** (Optional[Tuple[float, float]]) – maximum update in xy position of panels
- **max_z_step** (Optional[float]) – maximum update in z position of panels
- **max_xy_span_step** (Optional[Tuple[float, float]]) – maximum update in xy_span position of panels

on_backwards_end()

Records the current paramaters of each panel before they are updated.

Return type
None

on_step_end()

After the update step, goes through and hard-clamps parameter updates based on the difference between their current values and values before the update step.

Return type
None

class tomopt.optimisation.callbacks.detector_callbacks.**SigmoidPanelSmoothnessSchedule**(*smooth_range*)

Bases: *PostWarmupCallback*

Creates an annealing schedule for the smooth attribute of *SigmoidDetectorPanel*. This can be used to move from smooth, unphysical panel with high sensitivity outside the physical panel boundaries, to one with sharper decrease in resolution | efficiency at the edge, and so more closely resembles a physical panel, whilst still being differentiable.

Parameters
smooth_range (Tuple[float, float]) – tuple of initial and final values for the smooth attributes of all panels in the volume. A base-10 log schedule used over the number of epochs-total number of warmup epochs.

on_epoch_begin()

At the start of each training epoch, will anneal the *SigmoidDetectorPanel* s' smooth attributes, if the callback is active.

Return type
None

on_train_begin()

Sets all *SigmoidDetectorPanel* s to their initial smooth values.

Return type
None

tomopt.optimisation.callbacks.diagnostic_callbacks module

class tomopt.optimisation.callbacks.diagnostic_callbacks.**HitRecord**

Bases: *ScatterRecord*

Records the hits of the muons. Once recorded, the hits can be retrieved via the get_record() method. *plot_hit_density()* may be used to plot the hit record.

Warning: Currently this callback makes no distinction between different volume layouts, and is designed to used over a single volume layout.

TODO extend these to create one record per volume

on_scatter_end()

Saves the hits of the latest muon batch.

Return type
None

class tomopt.optimisation.callbacks.diagnostic_callbacks.**ScatterRecord**

Bases: *Callback*

Records the PoCAs of the muons which are located inside the passive volume. Once recorded, the PoCAs can be retrieved via the *get_record()* method. *plot_scatter_density()* may be used to plot the scatter record.

Warning: Currently this callback makes no distinction between different volume layouts, and is designed to used over a single volume layout.

TODO extend these to create one record per volume

get_record(*as_df=False*)

Access the recorded PoCAs.

Parameters
as_df (bool) – if True, will return a Pandas DataFrame, otherwise will return a Tensor

Return type
Union[Tensor, DataFrame]

Returns
a Pandas DataFrame or a Tensor of recorded PoCAs

`on_pred_begin()`

Prepares to record scatters

Return type
None

`on_scatter_end()`

Saves the PoCAs of the latest muon batch.

Return type
None

`on_train_begin()`

Prepares to record scatters

Return type
None

tomopt.optimisation.callbacks.eval_metric module

`class tomopt.optimisation.callbacks.eval_metric.EvalMetric(lower_metric_better, name=None, main_metric=True)`

Bases: *Callback*

Base class from which metric should inherit and implement the computation of their metric values. Inheriting classes will automatically be detected by *MetricLogger* and included in live feedback if it is the "main metric"

Parameters

- **lower_metric_better** (bool) – if True, a lower value of the metric should be considered better than a higher value
- **name** (Optional[str]) – name to associate with the metric
- **main_metric** (bool) – whether this metric should be considered the "main metric"

`get_metric()`

This will be called by *on_epoch_end()*

Return type
float

Returns
metric value

`on_train_begin()`

Ensures that only one main metric is used

Return type
None

tomopt.optimisation.callbacks.grad_callbacks module

class tomopt.optimisation.callbacks.grad_callbacks.**NoMoreNaNs**

Bases: *Callback*

Prior to parameter updates, this callback will check and set any NaN gradients to zero. Updates based on NaN gradients will set the parameter value to NaN.

Important: As new parameters are introduced, e.g. through new detector models, this callback will need to be updated.

---

on_backwards_end()

Prior to optimiser updates, parameter gradients are checked for NaNs.

Return type
None

tomopt.optimisation.callbacks.heatmap_gif module

class tomopt.optimisation.callbacks.heatmap_gif.**HeatMapGif**(*gif_filename='heatmap.gif'*)

Bases: *Callback*

Records a gif of the first heatmap in the first detector layer during training.

Parameters
gif_filename (str) – savename for the gif (will be appended to the callback savepath)

on_epoch_begin()

When a new training epoch begins, saves an image of the current layout of the first heatmap in the first detector layer

Return type
None

on_train_begin()

Prepares to record a new gif

Return type
None

on_train_end()

When training, saves an image of the current layout of the first heatmap in the first detector layer and then combines all images into a gif

Return type
None

tomopt.optimisation.callbacks.monitors module

class tomopt.optimisation.callbacks.monitors.**MetricLogger**(*gif_filename='optimisation_history.gif'*, *gif_length=10.0*, *show_plots=False*)

Bases: *Callback*

Provides live feedback during training showing a variety of metrics to help highlight problems or test hyper-parameters without completing a full training. If *show_plots* is false, will instead print training and validation losses at the end of each epoch. The full history is available as a dictionary by calling *get_loss_history()*. Additionally, a gif of the optimisation can be saved.

Parameters

- **gif_filename** (Optional[str]) – optional savename for recording a gif of the optimisation process (None -> no gif) The savename will be appended to the callback savepath
- **gif_length** (float) – If saving gifs, controls the total length in seconds
- **show_plots** (bool) – whether to provide live plots during optimisation in notebooks

cat_palette = 'tab10'

get_loss_history()

Get the current history of losses and metrics

Returns
tuple of ordered dictionaries: first with losses, second with validation metrics

Return type
history

get_results(*loaded_best*)

Return type
Dict[str, float]

h_mid = 8

lbl_col = 'black'

lbl_sz = 24

leg_sz = 16

on_backwards_end()

Records the training loss for the latest volume batch

Return type
None

on_epoch_begin()

Prepare to track new loss and snapshot the plots if training

Return type
None

on_epoch_end()

If validation epoch finished, record validation losses, compute info and update plots

Return type
None

on_train_begin()

Prepare for new training

Return type
None

on_train_end()

Cleans up plots, and optionally creates a gif of the training history

Return type
None

on_volume_batch_end()

Grabs the validation losses for the latest volume batch

Return type
None

on_volume_end()

Grabs the validation sub-losses for the latest volume

Return type
None

print_losses()

Print training and validation losses for the last epoch

Return type
None

style = {'rc': {'patch.edgecolor': 'none'}, 'style': 'whitegrid'}

tk_col = 'black'

tk_sz = 16

update_plot()

Updates the plot(s).

Return type
None

w_mid = 14.222222222222221

class tomopt.optimisation.callbacks.monitors.**PanelMetricLogger**(*gif_filename='optimisation_history.gif'*, *gif_length=10.0*, *show_plots=False*)

Bases: *MetricLogger*

Logger for use with *PanelDetectorLayer* s

Parameters

- **gif_filename** (Optional[str]) – optional savename for recording a gif of the optimisation process (None -> no gif) The savename will be appended to the callback savepath
- **gif_length** (float) – If saving gifs, controls the total length in seconds
- **show_plots** (bool) – whether to provide live plots during optimisation in notebooks

update_plot()

Updates the plot(s).

Return type
None

tomopt.optimisation.callbacks.opt_callbacks module

class tomopt.optimisation.callbacks.opt_callbacks.**EpochSave**

Bases: *Callback*

Saves the state of the volume at the end of each training epoch to a unique file. This can be used to load a specifc state to either be used, or to resume training.

on_epoch_end()

Runs when a training or validations epoch ends.

Return type
None

class tomopt.optimisation.callbacks.opt_callbacks.**OneCycle**(*opt_name*, *warmup_length*, *init_lr=None*, *init_mom=None*, *mid_lr=None*, *mid_mom=None*, *final_lr=None*, *final_mom=None*)

Bases: AbsOptSchedule

Callback implementing Smith 1-cycle evolution for lr and momentum (beta_1) https://arxiv.org/abs/1803.09820
In the warmup phase:
Learning rate is increased from *init_lr* to *mid_lr*, Momentum is decreased from *init_mom* to *mid_mom*, to stabilise the use of high LRs
In the convergence phase:
Learning rate is decreased from *mid_lr* to *final_lr*, Momentum is increased from *mid_mom* to *final_mom*
Setting the learning rate or momentum here will override the values specified when instantiating the *VolumeWrapper*. learning rate or momentum arguments can be *None* to avoid annealing or overriding their values.

Parameters

- **opt_name** (str) – name of optimiser that should be affected by the scheduler
- **warmup_length** (int) – number of epochs to use for the warmup phase
- **init_lr** (Optional[float]) – initial learning rate (low)
- **init_mom** (Optional[float]) – initial momentum (high)
- **mid_lr** (Optional[float]) – nominal learning rate (high),
- **mid_mom** (Optional[float]) – nominal momentum (moderate),
- **final_lr** (Optional[float]) – final learning rate (low),
- **final_mom** (Optional[float]) – final momentum (high)

on_epoch_end()

Runs when a training or validations epoch ends.

Return type
None

schedule()

Compute LR and momentum as a function of iter_cnt, according to defined ranges.

Return type
Tuple[Optional[float], Optional[float]]

tomopt.optimisation.callbacks.pred_callbacks module

class tomopt.optimisation.callbacks.pred_callbacks.**PredHandler**

Bases: *Callback*

Default callback for predictions. Collects predictions and true voxelwise X0 pairs for a range of volumes and returns them as list of tuples of numpy arrays when *get_preds()* is called.

get_preds()

Return type
List[Tuple[ndarray, ndarray]]

Returns
List of predicted and target pairs

on_pred_begin()

Prepares to record predictions

Return type
None

on_x0_pred_end()

Records predictions and true volume layout for the latest volume

Return type
None

class tomopt.optimisation.callbacks.pred_callbacks.**Save2HDF5PredHandler**(*path*, *use_volume_target*, *overwrite=False*, *x02id=None*, *compression='lzf'*)

Bases: *VolumeTargetPredHandler*

Saves predictions and targets to an HDF5 file, rather than caching and returning them. Samples are written incrementally. Can optionally save volume targets rather than voxel-wise X0 targets If an x02id lookup is provided, it transforms the target from an X0 value to a material class ID.

Parameters

- **path** (Path) – savename of file to save predictions and targets
- **use_volume_target** (bool) – if True, saves the volume target value instead of the volume X0s
- **overwrite** (bool) – if True will overwrite existing files with the same path, otherwise will append to them
- **x02id** (Optional[Dict[float, int]]) – optional map from X0 values to class IDs
- **compression** (Optional[str]) – optional string representation of any compression to use when saving data

on_x0_pred_end()

Records predictions and true volume layout or target for the latest volume

Return type
None

class tomopt.optimisation.callbacks.pred_callbacks.**VolumeTargetPredHandler**(*x02id=None*)

Bases: *PredHandler*

Returns the volume target as the target value, rather than the voxel-wise X0s. If an x02id lookup is provided, it transforms the target from an X0 value to a material class ID.

Parameters

x02id (Optional[Dict[float, int]]) – optional map from X0 values to class IDs

on_x0_pred_end()

Records predictions and volume target for the latest volume

Return type

None

tomopt.optimisation.callbacks.warmup_callbacks module

class tomopt.optimisation.callbacks.warmup_callbacks.**CostCoefWarmup**(*n_warmup*)

Bases: *WarmupCallback*

Sets a more stable cost coefficient in the *AbsDetectorLoss* by averaging the inference-error component for several epochs. During this warm-up monitoring phase, the detectors will be kept fixed.

Parameters

n_warmup (int) – number of training epochs to wait before setting the cost coefficient

on_epoch_end()

If enough epochs have past, the overall median inference-error is computed and used to set the cost coefficient in the loss.

Return type

None

on_volume_end()

If training, grabs the inference-error for the latest volume

Return type

None

class tomopt.optimisation.callbacks.warmup_callbacks.**OptConfig**(*n_warmup*, *rates*)

Bases: *WarmupCallback*

Allows the user to specify the desired update steps for parameters in physical units. Over the course of several warm-up epochs the gradients on the parameters are monitored, after which suitable learning rates for the optimisers are set, such that the parameters will move by the desired amount every update. During the warm-up, the detectors will not be updated as optimiser learning rates will be set to zero.

The calculation here does not account for the effect of the optimiser's momentum, nor scheduling and adaption of learning rates, and so the actual update rates may be different from the desired ones.

Parameters

- **n_warmup** (int) – number of training epochs to wait before setting learning rates
- **rates** (Dict[str, float]) – dictionary of desired update rates for the parameters The keys are the names of the optimisers specified in the optimiser dictionary of the wrapper. The values are the desired update rates for the parameters in physical units. For example, if the optimiser is SGD, and the parameter is the xy position of a panel, then the update rate should be in metres. The parameters that are being optimisered are expected to be found in the zeroth parameter group of the optimiser, i.e. *wrapper.opts[opt].param_groups[0]['params']*. This implies that the optimiser is expected to have only one parameter group.

Example::

```
>>> OptConfig(n_warmup=2, rates={'xy_pos_opt':xy_pos_rate, 'z_pos_opt':z_pos_
↪rate, 'xy_span_opt':xy_span_rate})
```

on_backwards_end()

Grabs training gradients from detector parameters

Return type
None

on_epoch_end()

When enough training epochs have passed, sets suitable learning rates for the optimisers based on the median gradients and desired update rates

Return type
None

class tomopt.optimisation.callbacks.warmup_callbacks.**PostWarmupCallback**

Bases: *Callback*

Callback class that waits for all *WarmupCallback* obejcts to finish their warmups before activating.

check_warmups()

When all WarmupCallbacks have finished, sets the callback to be active.

Return type
None

on_epoch_begin()

Checks to see whether the callback should be active.

Return type
None

on_train_begin()

Prepares for new training

Return type
None

class tomopt.optimisation.callbacks.warmup_callbacks.**WarmupCallback**(*n_warmup*)

Bases: *Callback*

Warmup callbacks act at the start of training to track and set parameters based on the initial state of the detector. During warmup, optimisation of the detector is prevented, via a flag. If multiple warmup callbacks are present, they will wait to warmup according to the order they are provided in. Once the last warmup callback finished, the flag will be set to allow the detectors to be optimised. When a *WarmupCallback* is warming up, its *warmup_active* attribute will be True.

Important: When inheriting from *WarmupCallback*, the super methods of *on_train_begin*, *on_epoch_begin*, and *on_epoch_end* must be called.

Parameters
n_warmup (int) – number of training epochs over-which to warmup

check_warmups()

If a *WarmupCallback* has finished, then its *warmup_active* is set to False, and the next *WarmupCallback* will have its *warmup_active* is set to True. If the finishing callback was the last *WarmupCallback*, then the "skip optimisation" flag is unset.

Return type
None

on_epoch_begin()

Ensures that when one *WarmupCallback* has finished, either the next is called, or the detectors are set to be optimised.

Return type
None

on_epoch_end()

After a training epoch is finished, increments the number of epochs that the callback has been warming up, provided it is active.

Return type
None

on_train_begin()

Prepares to warmup

Return type
None

tomopt.optimisation.data package

Submodules

tomopt.optimisation.data.passives module

class tomopt.optimisation.data.passives.**AbsBlockPassiveGenerator**(*volume*, *block_size*, *block_size_max_half=None*, *materials=None*)

Bases: *AbsPassiveGenerator*

Abstract base class for classes that generate new passive layouts which contain a single cuboid of material (block).
The _generate() method should be overridden to return:

- A function that provides an xy tensor for a given layer when called with its z position, length and width, and size.
- An optional "target" value for the layout

The *generate()* method will return only the layout function and no target The *get_data()* method will return both the layout function and the target

The block will be centred randomly in the volume, and can either be of fixed or random size.

Parameters

- **volume** (*Volume*) – Volume that the passive layout will be loaded into
- **block_size** (Optional[Tuple[float, float, float]]) – if set, will generate blocks of the specified size and random orientation, otherwise will randomly set the size of the blocks

- **block_size_max_half** (Optional[bool]) – if True and block_size is None, the maximum size of blocks will be set to half the size of the passive volume
- **materials** (Optional[List[str]]) – list of material names that can be used in the volume, None -> all materials known to TomOpt

class tomopt.optimisation.data.passives.**AbsPassiveGenerator**(*volume*, *materials=None*)

Bases: object

Abstract base class for classes that generate new passive layouts.
The _generate() method should be overridden to return:

- A function that provides an xy tensor for a given layer when called with its z position, length and width, and size.
- An optional "target" value for the layout

The *generate()* method will return only the layout function and no target The *get_data()* method will return both the layout function and the target

Parameters

- **volume** (*Volume*) – Volume that the passive layout will be loaded into
- **materials** (Optional[List[str]]) – list of material names that can be used in the volume, None -> all materials known to TomOpt

generate()

Return type
Callable[[Tensor, Tensor, float], Tensor]

Returns
The layout function and no target

get_data()

Returns
A function that provides an xy tensor for a given layer when called with its z position, length and width, and size. Target: An optional "target" value for the layout

Return type
RadLengthFunc

class tomopt.optimisation.data.passives.**BlockPresentPassiveGenerator**(*volume*, *block_size*, *block_size_max_half=None*, *materials=None*)

Bases: *AbsBlockPassiveGenerator*

Generates new passive layouts which contain a single cuboid of material (block) of random material against a fixed background material. Blocks are always present, but can potentially be of the same material as the background. The target for the volumes is the X0 of the block material. The background material for the background will always be the zeroth material provided during initialisation.

The *generate()* method will return only the layout function and no target The *get_data()* method will return both the layout function and the target

The block will be centred randomly in the volume, and can either be of fixed or random size.

Parameters

- **volume** (*Volume*) – Volume that the passive layout will be loaded into
- **block_size** (Optional[Tuple[float, float, float]]) – if set, will generate blocks of the specified size and random orientation, otherwise will randomly set the size of the blocks

- **block_size_max_half** (Optional[bool]) – if True and block_size is None, the maximum size of blocks will be set to half the size of the passive volume
- **materials** (Optional[List[str]]) – list of material names that can be used in the volume, None -> all materials known to TomOpt

class tomopt.optimisation.data.passives.**PassiveYielder**(*passives*, *n_passives=None*, *shuffle=True*)

Bases: object

Dataset class that can either:

Yield from a set of pre-specified passive-volume layouts, and optional targets Generate and yield random layouts and optional targets from a provided generator

Parameters

- **passives** (Union[List[Union[Tuple[Callable[[Tensor, Tensor, float], Tensor], Optional[Tensor]], Callable[[Tensor, Tensor, float], Tensor]]], *AbsPassiveGenerator*]) – Either a list of passive-volume functions (and optional targets together in a tuple), or a passive-volume generator
- **n_passives** (Optional[int]) – if a generator is used, this determines the number of volumes to generator per epoch in training, or in total when predicting
- **shuffle** (bool) – If a list of pre-specified layouts is provided, their order will be shuffled if this is True

class tomopt.optimisation.data.passives.**RandomBlockPassiveGenerator**(*volume*, *block_size*, *sort_x0*, *enforce_diff_mat*, *block_size_max_half=None*, *materials=None*)

Bases: *AbsBlockPassiveGenerator*

Generates new passive layouts which contain a single cuboid of material (block) of random material against a random background material. Blocks are always present, but can potentially be of the same material as the background. The target for the volumes is the X0 of the block material.

The *generate()* method will return only the layout function and no target The *get_data()* method will return both the layout function and the target

The block will be centred randomly in the volume, and can either be of fixed or random size.

Parameters

- **volume** (*Volume*) – Volume that the passive layout will be loaded into
- **block_size** (Optional[Tuple[float, float, float]]) – if set, will generate blocks of the specified size and random orientation, otherwise will randomly set the size of the blocks
- **sort_x0** (bool) – if True, the block will always have a lower X0 than the background, unless they are of the same material
- **enforce_diff_mat** (bool) – if True, the block will always be of a different material to the background
- **block_size_max_half** (Optional[bool]) – if True and block_size is None, the maximum size of blocks will be set to half the size of the passive volume
- **materials** (Optional[List[str]]) – list of material names that can be used in the volume, None -> all materials known to TomOpt

class tomopt.optimisation.data.passives.**VoxelPassiveGenerator**(*volume*, *materials=None*)

Bases: *AbsPassiveGenerator*

Generates new passive layouts where every voxel is of a random material.

The *generate()* method will return only the layout function and no target The *get_data()* method will return both the layout function and the target

Parameters

- **volume** (*Volume*) – Volume that the passive layout will be loaded into
- **materials** (Optional[List[str]]) – list of material names that can be used in the volume, None -> all materials known to TomOpt

tomopt.optimisation.loss package

Submodules

tomopt.optimisation.loss.loss module

class tomopt.optimisation.loss.loss.**AbsDetectorLoss**(*, *target_budget*, *budget_smoothing=10*, *cost_coef=None*, *steep_budget=True*, *debug=False*)

Bases: Module

Abstract base class from which all loss functions should inherit.
The loss consists of:

- A component that quantifies the performance of the predictions made via the detectors
- An optional component that relates to the cost of the detector

The total loss is the sum of these, with the cost-component being rescaled by a coefficient characterising its relative importance.

The performance component (error) should ideally be as close to the final task that the detector will be performing, and will depend on the output of the inference algorithm used

The optional cost component is included as a budget weighting, which gradually increases with the current cost up to a predefined budget, after which it increases rapidly, but smoothly. Be default, the budget is based on a sigmoid centred at the budget, which linearly increases after the budget is exceeded. A less steep version is selectable, which flattens out slightly for high costs.

Inheriting classes will need to at least override the *_get_inference_loss* method.

Parameters

- **target_budget** (Optional[float]) – If not None, will include a cost component in the loss configured for the specified budget. Should be specified in the same currency units as the detector cost.
- **budget_smoothing** (float) – controls how quickly the budget term rises with cost; lower values => slower rise
- **cost_coef** (Union[Tensor, float, None]) – Balancing coefficient used to multiply the budget term prior to its addition to the error component of the loss. If set to None, it will be set equal to the inference-error computed the first time the loss is computed
- **steep_budget** (bool) – If True, will use a linearly increasing budget term when the budget is exceeded, otherwise the budget term will flatten off for very high costs
- **debug** (bool) – If True, will print out information about the loss whenever it is evaluated

forward(*pred*, *volume*)

Computes the loss for the predictions of a single volume using the current state of the detector

Parameters

- **pred** (Tensor) – the predictions from the inference
- **volume** (*Volume*) – Volume containing the passive volume that was being predicted and the detector being optimised

Return type
Tensor

Returns
The loss for the predictions and detector

class tomopt.optimisation.loss.loss.**AbsMaterialClassLoss**(*, *x02id*, *target_budget*, *budget_smoothing=10*, *cost_coef=None*, *steep_budget=True*, *debug=False*)

Bases: *AbsDetectorLoss*

Abstract base class for cases in which the task is to classify materials in the passive volumes, or some other aspect of the volumes. The targets returned by the volume are expected to be float X0s, and are converted to class IDs using an X0 to ID map.
The loss consists of:

- A component that quantifies the performance of the predictions made via the detectors
- An optional component that relates to the cost of the detector

The total loss is the sum of these, with the cost-component being rescaled by a coefficient characterising its relative importance.

The performance component (error) should ideally be as close to the final task that the detector will be performing, and will depend on the output of the inference algorithm used

The optional cost component is included as a budget weighting, which gradually increases with the current cost up to a predefined budget, after which it increases rapidly, but smoothly. Be default, the budget is based on a sigmoid centred at the budget, which linearly increases after the budget is exceeded. A less steep version is selectable, which flattens out slightly for high costs.

Inheriting classes will need to at least override the *_get_inference_loss* method.

Parameters

- **x02id** (Dict[float, int]) – Dictionary mapping float X0 targets to integer class IDs
- **target_budget** (float) – If not None, will include a cost component in the loss configured for the specified budget. Should be specified in the same currency units as the detector cost.
- **budget_smoothing** (float) – controls how quickly the budget term rises with cost; lower values => slower rise
- **cost_coef** (Union[Tensor, float, None]) – Balancing coefficient used to multiply the budget term prior to its addition to the error component of the loss. If set to None, it will be set equal to the inference-error computed the first time the loss is computed
- **steep_budget** (bool) – If True, will use a linearly increasing budget term when the budget is exceeded, otherwise the budget term will flatten off for very high costs
- **debug** (bool) – If True, will print out information about the loss whenever it is evaluated

class tomopt.optimisation.loss.loss.**VolumeClassLoss**(*, *x02id*, *target_budget*, *budget_smoothing=10*, *cost_coef=None*, *steep_budget=True*, *debug=False*)

Bases: *AbsMaterialClassLoss*

Loss function designed for tasks where some overall target of the passive volume must be classified, and the target of the volume is encoded as a float X0. E.g. what is the material of a large block in the volume.

The Inference-error component of the loss depends on shape of predictions provided:

If the predictions are of shape (1,classes,voxels), they will be interpreted as multi-class log-probabilities and the negative log-likelihood computed If the predictions are of shape (1,1,voxels), they will be interpreted as binary class probabilities and the binary cross-entropy computed

The ordering of the "flattened" voxels should match that of *volume.get_rad_cube().flatten()*

The total loss consists of:

- The NLL or BCE
- An optional component that relates to the cost of the detector

The total loss is the sum of these, with the cost-component being rescaled by a coefficient characterising its relative importance.

The optional cost component is included as a budget weighting, which gradually increases with the current cost up to a predefined budget, after which it increases rapidly, but smoothly. Be default, the budget is based on a sigmoid centred at the budget, which linearly increases after the budget is exceeded. A less steep version is selectable, which flattens out slightly for high costs.

Parameters

- **x02id** (Dict[float, int]) – Dictionary mapping float X0 targets to integer class IDs
- **target_budget** (float) – If not None, will include a cost component in the loss configured for the specified budget. Should be specified in the same currency units as the detector cost.
- **budget_smoothing** (float) – controls how quickly the budget term rises with cost; lower values => slower rise
- **cost_coef** (Union[Tensor, float, None]) – Balancing coefficient used to multiply the budget term prior to its addition to the error component of the loss. If set to None, it will be set equal to the inference-error computed the first time the loss is computed
- **steep_budget** (bool) – If True, will use a linearly increasing budget term when the budget is exceeded, otherwise the budget term will flatten off for very high costs
- **debug** (bool) – If True, will print out information about the loss whenever it is evaluated

class tomopt.optimisation.loss.loss.**VolumeIntClassLoss**(*, *targ2int*, *pred_int_start*, *use_mse*, *target_budget*, *budget_smoothing=10*, *cost_coef=None*, *steep_budget=True*, *debug=False*)

Bases: *AbsDetectorLoss*

Loss function designed for tasks where some overall integer target of the passive volume must be classified, and the values of this target are quantifiably comparable (i.e. the integers are treatable as numbers not just categorical codes). E.g. Predicting how many layers of the passive volume are filled with a given material.

The Inference-error component of the loss computed as the *integer_class_loss()*. Predictions should be provided as probabilities for every possible integer target The target from the volume can be converted to an integer (e.g. height to layer ID) using a *targ2int* function

The total loss consists of:

- The integer class loss (ICL)

- An optional component that relates to the cost of the detector

The total loss is the sum of these, with the cost-component being rescaled by a coefficient characterising its relative importance.

The optional cost component is included as a budget weighting, which gradually increases with the current cost up to a predefined budget, after which it increases rapidly, but smoothly. Be default, the budget is based on a sigmoid centred at the budget, which linearly increases after the budget is exceeded. A less steep version is selectable, which flattens out slightly for high costs.

Parameters

- **target_budget** (`float`) – If not None, will include a cost component in the loss configured for the specified budget. Should be specified in the same currency units as the detector cost.
- **budget_smoothing** (`float`) – controls how quickly the budget term rises with cost; lower values => slower rise
- **cost_coef** (`Union[Tensor, float, None]`) – Balancing coefficient used to multiply the budget term prior to its addition to the error component of the loss. If set to None, it will be set equal to the inference-error computed the first time the loss is computed
- **steep_budget** (`bool`) – If True, will use a linearly increasing budget term when the budget is exceeded, otherwise the budget term will flatten off for very high costs
- **debug** (`bool`) – If True, will print out information about the loss whenever it is evaluated

class tomopt.optimisation.loss.loss.**VolumeMSELoss**(*, *target_budget*, *budget_smoothing=10*, *cost_coef=None*, *steep_budget=True*, *debug=False*)

Bases: *AbsDetectorLoss*

TODO: Add unit tests and docs

class tomopt.optimisation.loss.loss.**VoxelClassLoss**(*, *x02id*, *target_budget*, *budget_smoothing=10*, *cost_coef=None*, *steep_budget=True*, *debug=False*)

Bases: *AbsMaterialClassLoss*

Loss function designed for tasks where the voxelwise material class ID must be classified. Inference-error component of the loss is the negative log-likelihood on log class-probabilities, averaged over all voxels (NLL)

Predictions should be provided as log-softmaxed class probabilities per voxel, with shape (1,classes,voxels). The ordering of the "flattened" voxels should match that of *volume.get_rad_cube().flatten()*

The total loss consists of:

- The NLL
- An optional component that relates to the cost of the detector

The total loss is the sum of these, with the cost-component being rescaled by a coefficient characterising its relative importance.

The optional cost component is included as a budget weighting, which gradually increases with the current cost up to a predefined budget, after which it increases rapidly, but smoothly. Be default, the budget is based on a sigmoid centred at the budget, which linearly increases after the budget is exceeded. A less steep version is selectable, which flattens out slightly for high costs.

Parameters

- **x02id** (`Dict[float, int]`) – Dictionary mapping float X0 targets to integer class IDs

- **target_budget** (float) – If not None, will include a cost component in the loss configured for the specified budget. Should be specified in the same currency units as the detector cost.
- **budget_smoothing** (float) – controls how quickly the budget term rises with cost; lower values => slower rise
- **cost_coef** (Union[Tensor, float, None]) – Balancing coefficient used to multiply the budget term prior to its addition to the error component of the loss. If set to None, it will be set equal to the inference-error computed the first time the loss is computed
- **steep_budget** (bool) – If True, will use a linearly increasing budget term when the budget is exceeded, otherwise the budget term will flatten off for very high costs
- **debug** (bool) – If True, will print out information about the loss whenever it is evaluated

class tomopt.optimisation.loss.loss.**VoxelX0Loss**(*, *target_budget*, *budget_smoothing=10*, *cost_coef=None*, *steep_budget=True*, *debug=False*)

Bases: *AbsDetectorLoss*

Loss function designed for tasks where the voxelwise X0 value must be predicted as floats. Inference-error component of the loss is the squared-error on X0 predictions, averaged over all voxels (MSE)
The total loss consists of:

- The MSE
- An optional component that relates to the cost of the detector

The total loss is the sum of these, with the cost-component being rescaled by a coefficient characterising its relative importance.

The optional cost component is included as a budget weighting, which gradually increases with the current cost up to a predefined budget, after which it increases rapidly, but smoothly. Be default, the budget is based on a sigmoid centred at the budget, which linearly increases after the budget is exceeded. A less steep version is selectable, which flattens out slightly for high costs.

Parameters

- **target_budget** (Optional[float]) – If not None, will include a cost component in the loss configured for the specified budget. Should be specified in the same currency units as the detector cost.
- **budget_smoothing** (float) – controls how quickly the budget term rises with cost; lower values => slower rise
- **cost_coef** (Union[Tensor, float, None]) – Balancing coefficient used to multiply the budget term prior to its addition to the error component of the loss. If set to None, it will be set equal to the inference-error computed the first time the loss is computed
- **steep_budget** (bool) – If True, will use a linearly increasing budget term when the budget is exceeded, otherwise the budget term will flatten off for very high costs
- **debug** (bool) – If True, will print out information about the loss whenever it is evaluated

tomopt.optimisation.loss.sub_losses module

tomopt.optimisation.loss.sub_losses.**integer_class_loss**(*int_probs*, *target_int*, *pred_start_int*, *use_mse*, *weight=None*, *reduction='mean'*)

Loss for classifying integers, when regression is not applicable. It assumed that the the integers really are quantifiably comparable, and not categorical codes of classes.

Like multiclass-classification, predictions are a probabilities for each possible integer, but the ICL aims to penalise close predictions less than far-off ones: For a target of 3 and a close prediction of *softmax([1,3,10,5,5,3,1])* and a far-off prediction of *softmax([10,3,1,5,5,3,1])*, the categorical cross-entropy produces the same loss for both predictions (5.0154) despite the close prediction having a higher probability near the target.

ICL instead computes the absolute error, or squared error, between each of the possible integers and the true target. These errors are then normalised, weighted by the predicted probabilities, and summed. I.e. integers close to the target have a lower error, and these are given greater weight in the sum if they have a higher probability.

For the example, the ICL produces a loss of 1.0007 for the close prediction, and 8.8773 for the far-off one.

Parameters

- **int_probs** (Tensor) – (*,integers) tensor of predicted probabilities
- **target_int** (Tensor) – (*) tensor of target integers
- **pred_start_int** (int) – the integer that the zeroth probability in predictions corresponds to
- **use_mse** (bool) – whether to compute errors as absolute or squared
- **weight** (Optional[Tensor]) – Optional (*) tensor of multiplicative weights for the unreduced ICLs
- **reduction** (str) – ‘mean’ return the average ICL, ‘sum’ sum the ICLs, ‘none’, return the individual ICLs

Return type

Tensor

tomopt.optimisation.wrapper package

Submodules

tomopt.optimisation.wrapper.volume_wrapper module

class tomopt.optimisation.wrapper.volume_wrapper.**AbsVolumeWrapper**(*volume*, ***, *partial_opts*, *loss_func=None*, *partial_scatter_inferrer*, *partial_volume_inferrer*, *mu_generator=None*)

Bases: object

Abstract base class for optimisation volume wrappers. Inheriting classes will need to override _build_opt() according to the detector parameters that need to be optimised.

Volume wrappers are designed to contain a *Volume* and provide means of optimising the detectors it contains, via their *fit()* method.

Wrappers also provide for various quality-of-life methods, such as saving and loading detector configurations, and computing predictions with a fixed detector (*predict()*)

Fitting of a detector proceeds as training and validation epochs, each of which contains multiple batches of passive volumes. For each volume in a batch, the loss is evaluated using multiple batches of muons. The whole loop is:

1. **for epoch in *n_epochs*:**
    A. *loss* = 0
    B. **for *p*, *passive* in enumerate(*trn_passives*):**
        a. load *passive* into passive volume
        b. **for muon_batch in range(*n_mu_per_volume*//*mu_bs*):**
            i. Irradiate volume with *mu_bs* muons
            ii. Infer passive volume
        c. Compute loss based on precision and cost, and add to *loss*
        d. **if *p* \`+1 % \`*passive_bs* == 0:**
            i. *loss* = *loss*/*passive_bs*
            ii. Backpropagate *loss* and update detector parameters
            iii. *loss* = 0
        e. **if len(*trn_passives*)-(*p* \`+1) < \`*passive_bs*:**
            i. Break
    C. *val_loss* = 0
    D. **for *p*, *passive* in enumerate(*val_passives*):**
        a. load *passive* into passive volume
        b. **for muon_batch in range(*n_mu_per_volume*//*mu_bs*):**
            i. Irradiate volume with *mu_bs* muons
            ii. Infer passive volume
            iii. Compute loss based on precision and cost, and add to *val_loss*
        c. **if len(*val_passives*)-(*p* \`+1) < \`*passive_bs*:**
            i. Break
    E. *val_loss* = *val_loss*/*p*

In implementation, the loop is broken up into several functions:

- `_fit_epoch()` runs one full epoch of volumes and updates for both training and validation
- `_scan_volumes()` runs over all training/validation volumes, updating parameters when necessary
- `_scan_volume()` irradiates a single volume with muons multiple batches, and computes the loss for that volume

The optimisation and prediction loops are supported by a stateful callback mechanism. The base callback is `Callback`, which can interject at various points in the loops. All aspects of the optimisation and prediction are stored in a `FitParams` data class, since the callbacks are also stored there, and the callbacks have a reference to the wrapper, they are able to read/write to the *FitParams* and be aware of other callbacks that are running.

Accounting for the interjection calls (*on_*_begin* & *on_*_end*), the full optimisation loop is:

1. Associate callbacks with wrapper (*set_wrapper*)
2. *on_train_begin*

3. **for epoch in *n_epochs*:**
    A. *state* = "train"
    B. *on_epoch_begin*
    C. **for *p*, *passive* in enumerate(*trn_passives*):**
        a. **if *p* % *passive_bs* == 0:**
            i. *on_volume_batch_begin*
            ii. *loss* = 0
        b. load *passive* into passive volume
        c. *on_volume_begin*
        d. **for muon_batch in range(*n_mu_per_volume*//*mu_bs*):**
            i. *on_mu_batch_begin*
            ii. Irradiate volume with *mu_bs* muons
            iii. Infer scatter locations
            iv. *on_scatter_end*
            v. Infer x0 and append to list of x0 predictions
            vi. *on_mu_batch_end*
        e. *on_x0_pred_begin*
        f. Compute overall x0 prediction
        g. *on_x0_pred_end*
        h. Compute loss based on precision and cost, and add to *loss*
        i. **if *p* `+1 % `passive_bs* == 0:**
            i. *loss* = *loss*/*passive_bs*
            ii. *on_volume_batch_end*
            iii. Zero parameter gradients
            iv. *on_backwards_begin*
            v. Backpropagate *loss* and compute parameter gradients
            vi. *on_backwards_end*
            vii. Update detector parameters
            viii. Ensure detector parameters are within physical boundaries (*AbsDetectorLayer.conform_detector*) viv. *loss* = 0
        j. **if len(*trn_passives*)-(*p* `+1) < `passive_bs*:**
            i. Break
    D. *on_epoch_end*
    E. *state* = "valid"
    F. *on_epoch_begin*
    G. **for *p*, *passive* in enumerate(*val_passives*):**

a. **if** ***p*** **%** ***passive_bs*** **== 0:**
   i. *on_volume_batch_begin*
   ii. *loss* = 0
b. *on_volume_begin*
c. **for muon_batch in range(*n_mu_per_volume*//*mu_bs*):**
   i. *on_mu_batch_begin*
   ii. Irradiate volume with *mu_bs* muons
   iii. Infer scatter locations
   iv. *on_scatter_end*
   v. Infer x0 and append to list of x0 predictions
   vi. *on_mu_batch_end*
d. *on_x0_pred_begin*
e. Compute overall x0 prediction
f. *on_x0_pred_end*
g. Compute loss based on precision and cost, and add to *loss*
h. **if** ***p*** **`+1 %** ***`passive_bs*** **== 0:**
   i. *loss* = *loss*/*passive_bs*
   ii. *on_volume_batch_end*
i. **if len(*val_passives*)-(*p* `+1) <** ***`passive_bs*****:**
   i. Break

H. *on_epoch_end*

4. *on_train_end*

Parameters

- **volume** (*Volume*) – the volume containing the detectors to be optimised
- **partial_opts** (Dict[str, Callable[[Iterator[Parameter]], Optimizer]]) – dictionary of uninitialised optimisers to be associated with the detector parameters, via *_build_opt*
- **loss_func** (Optional[*AbsDetectorLoss*]) – Optional loss function (required if planning to optimise the detectors)
- **partial_scatter_inferrer** (Type[*ScatterBatch*]) – uninitialised class to be used for inferring muon scatter variables and trajectories
- **partial_volume_inferrer** (Type[*AbsVolumeInferrer*]) – uninitialised class to be used for inferring volume targets
- **mu_generator** (Optional[*AbsMuonGenerator*]) – Optional generator class for muons. If None, will use `from_volume()`.

fit(*n_epochs*, *passive_bs*, *n_mu_per_volume*, *mu_bs*, *trn_passives*, *val_passives*, *cbs=None*, *cb_savepath=Path('train_weights')*)

Runs the fitting loop for the detectors over a specified number of epochs, using the provided volumes or volume generators. The optimisation loop can be supported by callbacks.

Parameters

- **n_epochs** (int) – number of epochs to run for (a training and validation epoch only counts as one 'epoch)
- **passive_bs** (int) – number of passive volumes to use per volume batch (detector updates occur after every volume batch in training mode)
- **n_mu_per_volume** (int) – number of muons to use in total when inferring the target of a single volume
- **mu_bs** (int) – number of muons to use per muon batch; multiple muon batches will be used until *n_mu_per_volume* is reached
- **trn_passives** (*PassiveYielder*) – passive volumes to use for optimising the detector
- **val_passives** (Optional[*PassiveYielder*]) – optional passive volumes to use for evaluating the detector
- **cbs** (Optional[List[*Callback*]]) – optional list of callbacks to use
- **cb_savepath** (Path) – location where callbacks should write/save any information

Return type
List[*Callback*]

Returns
The list of callbacks

get_detectors()

Return type
List[*AbsDetectorLayer*]

Returns
A list of all *AbsDetectorLayer* s in the volume, in the order of *layers* (normally decreasing z position)

get_opt_lr(*opt*)

Returns the learning rate of the specified optimiser.

Parameters
opt (str) – string name of the optimiser requested

Return type
float

Returns
The learning rate of the specified optimiser

get_opt_mom(*opt*)

Returns the momentum coefficient/beta_1 of the specified optimiser.

Parameters
opt (str) – string name of the optimiser requested

Return type
float

Returns
The momentum coefficient/beta_1 of the specified optimiser

get_param_count(*trainable=True*)

Return number of parameters in detector.

Parameters
trainable (bool) – if true (default) only count trainable parameters

Return type
int

Returns
Number of (trainable) parameters in detector

load(*name*)

Loads saved volume and optimiser parameters from a file.

Parameters
name (str) – file to load

Return type
None

opts: Dict[str, Optimizer]

predict(*passives*, *n_mu_per_volume*, *mu_bs*, *pred_cb=<tomopt.optimisation.callbacks.pred_callbacks.PredHandler object>*, *cbs=None*, *cb_savepath=Path('train_weights')*)

Uses the detectors to predict the provided volumes The prediction loop can be supported by callbacks.

Parameters

- **passives** (*PassiveYielder*) – passive volumes to predict
- **n_mu_per_volume** (int) – number of muons to use in total when inferring the target of a single volume
- **mu_bs** (int) – number of muons to use per muon batch; multiple muon batches will be used until n_mu_per_volume is reached
- **pred_cb** (*PredHandler*) – the prediction callback to use for recording predictions
- **cbs** (Optional[List[*Callback*]]) – optional list of callbacks to use
- **cb_savepath** (Path) – location where callbacks should write/save any information

Return type
List[Tuple[ndarray, ndarray]]

Returns
The object returned by the *pred_cb*'s *get_preds* method

save(*name*)

Saves the volume and optimiser parameters to a file.

Parameters
name (str) – savename for the file

Return type
None

set_opt_lr(*lr*, *opt*)

Sets the learning rate of the specified optimiser.

Parameters

- **lr** (float) – new learning rate for the optimiser
- **opt** (str) – string name of the optimiser requested

Return type
None

set_opt_mom(*mom*, *opt*)

Sets the learning rate of the specified optimiser.

Parameters

- **mom** (float) – new momentum coefficient/beta_1 for the optimiser
- **opt** (str) – string name of the optimiser requested

Return type
None

class tomopt.optimisation.wrapper.volume_wrapper.**ArbVolumeWrapper**(*volume*, ***, *opts*, *loss_func=None*, *partial_scatter_inferrer=<class 'tomopt.inference.scattering.ScatterBatch'>*, *partial_volume_inferrer=<class 'tomopt.inference.volume.PanelX0Inferrer'>*, *mu_generator=None*)

Bases: *AbsVolumeWrapper*

Arbitrary volume wrapper in which the user supplies pre-instantiated optimisers for whatever paramters should be optimised.

Wrappers also provide for various quality-of-life methods, such as saving and loading detector configurations, and computing predictions with a fixed detector (*predict()*)

Fitting of a detector proceeds as training and validation epochs, each of which contains multiple batches of passive volumes. For each volume in a batch, the loss is evaluated using multiple batches of muons. The whole loop is:

1. **for epoch in *n_epochs*:**
    A. *loss* = 0
    B. **for *p*, *passive* in enumerate(*trn_passives*):**
        a. load *passive* into passive volume
        b. **for muon_batch in range(*n_mu_per_volume*//*mu_bs*):**
            i. Irradiate volume with *mu_bs* muons
            ii. Infer passive volume
        c. Compute loss based on precision and cost, and add to *loss*
        d. **if *p* `+1 % `passive_bs == 0:**
            i. *loss* = *loss*/*passive_bs*
            ii. Backpropagate *loss* and update detector parameters
            iii. *loss* = 0
        e. **if len(*trn_passives*)-(*p* `+1) < `passive_bs:**

i. Break

C. *val_loss* = 0

D. **for *p*, *passive* in enumerate(*val_passives*):**

a. load *passive* into passive volume

b. **for muon_batch in range(*n_mu_per_volume*//*mu_bs*):**

i. Irradiate volume with *mu_bs* muons

ii. Infer passive volume

iii. Compute loss based on precision and cost, and add to *val_loss*

c. **if len(*val_passives*)-(*p* `+1) < `*passive_bs*:**

i. Break

E. *val_loss* = *val_loss*/*p*

In implementation, the loop is broken up into several functions:

`_fit_epoch()` runs one full epoch of volumes
and updates for both training and validation

`_scan_volumes()` runs over all training/validation volumes,
updating parameters when necessary

`_scan_volume()` irradiates a single volume with muons multiple batches,
and computes the loss for that volume

The optimisation and prediction loops are supported by a stateful callback mechanism. The base callback is `Callback`, which can interject at various points in the loops. All aspects of the optimisation and prediction are stored in a `FitParams` data class, since the callbacks are also stored there, and the callbacks have a reference to the wrapper, they are able to read/write to the *FitParams* and be aware of other callbacks that are running.

Accounting for the interjection calls (*on_*_begin* & *on_*_end*), the full optimisation loop is:

1. Associate callbacks with wrapper (*set_wrapper*)
2. *on_train_begin*
3. **for epoch in *n_epochs*:**

A. *state* = "train"

B. *on_epoch_begin*

C. **for *p*, *passive* in enumerate(*trn_passives*):**

a. **if *p* % *passive_bs* == 0:**

i. *on_volume_batch_begin*

ii. *loss* = 0

b. load *passive* into passive volume

c. *on_volume_begin*

d. **for muon_batch in range(*n_mu_per_volume*//*mu_bs*):**

i. *on_mu_batch_begin*

ii. Irradiate volume with *mu_bs* muons

iii. Infer scatter locations

iv. *on_scatter_end*

v. Infer x0 and append to list of x0 predictions

vi. *on_mu_batch_end*

e. *on_x0_pred_begin*

f. Compute overall x0 prediction

g. *on_x0_pred_end*

h. Compute loss based on precision and cost, and add to *loss*

i. **if *p* `+1 % `*passive_bs* == 0:**

i. *loss* = *loss*/*passive_bs*

ii. *on_volume_batch_end*

iii. Zero parameter gradients

iv. *on_backwards_begin*

v. Backpropagate *loss* and compute parameter gradients

vi. *on_backwards_end*

vii. Update detector parameters

viii. Ensure detector parameters are within physical boundaries (*AbsDetectorLayer.conform_detector*) viv. *loss* = 0

j. **if len(*trn_passives*)-(*p* `+1) < `*passive_bs*:**

i. Break

D. *on_epoch_end*

E. *state* = "valid"

F. *on_epoch_begin*

G. **for *p*, *passive* in enumerate(*val_passives*):**

a. **if *p* % *passive_bs* == 0:**

i. *on_volume_batch_begin*

ii. *loss* = 0

b. *on_volume_begin*

c. **for muon_batch in range(*n_mu_per_volume*//*mu_bs*):**

i. *on_mu_batch_begin*

ii. Irradiate volume with *mu_bs* muons

iii. Infer scatter locations

iv. *on_scatter_end*

v. Infer x0 and append to list of x0 predictions

vi. *on_mu_batch_end*

d. *on_x0_pred_begin*

e. Compute overall x0 prediction

f. *on_x0_pred_end*

g. Compute loss based on precision and cost, and add to *loss*

h. **if *p`+1 % `passive_bs* == 0:**

i. *loss = loss/passive_bs*

ii. *on_volume_batch_end*

i. **if len(*val_passives*)-(*p`+1*) < *`passive_bs*:**

i. Break

H. *on_epoch_end*

4. *on_train_end*

Parameters

- **volume** (*Volume*) – the volume containing the detectors to be optimised
- **opts** (Dict[str, Optimizer]) – Dict of strings mapping to initialised optimisers
- **loss_func** (Optional[*AbsDetectorLoss*]) – optional loss function (required if planning to optimise the detectors)
- **partial_scatter_inferrer** (Type[*ScatterBatch*]) – uninitialised class to be used for inferring muon scatter variables and trajectories
- **partial_volume_inferrer** (Type[*AbsVolumeInferrer*]) – uninitialised class to be used for inferring volume targets
- **mu_generator** (Optional[*AbsMuonGenerator*]) – Optional generator class for muons. If None, will use `from_volume()`.

classmethod from_save(*name*, ***, *volume*, *opts*, *loss_func*, *partial_scatter_inferrer=<class 'tomopt.inference.scattering.ScatterBatch'>*, *partial_volume_inferrer=<class 'tomopt.inference.volume.PanelX0Inferrer'>*, *mu_generator=None*)

Instantiates a new *PanelVolumeWrapper* and loads saved detector and optimiser parameters

Parameters

- **name** (str) – file name with saved detector and optimiser parameters
- **volume** (*Volume*) – the volume containing the detectors to be optimised
- **opts** (Dict[str, Optimizer]) – Dict of strings mapping to initialised optimisers
- **loss_func** (Optional[*AbsDetectorLoss*]) – optional loss function (required if planning to optimise the detectors)
- **partial_scatter_inferrer** (Type[*ScatterBatch*]) – uninitialised class to be used for inferring muon scatter variables and trajectories
- **partial_volume_inferrer** (Type[*AbsVolumeInferrer*]) – uninitialised class to be used for inferring volume targets
- **mu_generator** (Optional[*AbsMuonGenerator*]) – Optional generator class for muons. If None, will use `from_volume()`.

Return type

AbsVolumeWrapper

class tomopt.optimisation.wrapper.volume_wrapper.**FitParams**(***kwargs*)

Bases: object

Data class used for storing all aspects of optimisation and prediction when working with *AbsVolumeWrapper*

Parameters

kwargs (Any) – objects to be stored

cb_savepath: Optional[Path] = None

cbs: Optional[List[*Callback*]] = None

cyclic_cbs: Optional[List[*CyclicCallback*]] = None

device: device = device(type='cpu')

epoch: int = 0

epoch_bar: Optional[ProgressBar] = None

loss_val: Optional[Tensor] = None

mean_loss: Optional[Tensor] = None

metric_cbs: Optional[List[*EvalMetric*]] = None

metric_log: Optional[*MetricLogger*] = None

mu: Optional[*MuonBatch*] = None

mu_bs: Optional[int] = None

n_epochs: Optional[int] = None

n_mu_per_volume: Optional[int] = None

passive_bar: Union[NBProgressBar, ConsoleProgressBar, None] = None

passive_bs: Optional[int] = None

pred: Optional[Tensor] = None

sb: Optional[*ScatterBatch*] = None

skip_opt_step: bool = False

state: Optional[str] = None

stop: Optional[bool] = None

trn_passives: Optional[*PassiveYielder*] = None

tst_passives: Optional[*PassiveYielder*] = None

val_passives: Optional[*PassiveYielder*] = None

volume_id: Optional[int] = None

volume_inferrer: Optional[*AbsVolumeInferrer*] = None

warmup_cbs: Optional[List[*WarmupCallback*]] = None

class tomopt.optimisation.wrapper.volume_wrapper.**HeatMapVolumeWrapper**(*volume*, *, *mu_opt*, *norm_opt*, *sig_opt*, *z_pos_opt*, *loss_func*, *partial_scatter_inferrer=<class 'tomopt.inference.scattering.ScatterBatch'>*, *partial_volume_inferrer=<class 'tomopt.inference.volume.PanelX0Inferrer'>*, *mu_generator=None*)

Bases: *AbsVolumeWrapper*

Volume wrapper for volumes with *DetectorHeatMap*-based detectors.

Volume wrappers are designed to contain a *Volume* and provide means of optimising the detectors it contains, via their *fit()* method.

Wrappers also provide for various quality-of-life methods, such as saving and loading detector configurations, and computing predictions with a fixed detector (*predict()*)

Fitting of a detector proceeds as training and validation epochs, each of which contains multiple batches of passive volumes. For each volume in a batch, the loss is evaluated using multiple batches of muons. The whole loop is:

1. **for epoch in *n_epochs*:**
    A. *loss* = 0
    B. **for *p*, *passive* in enumerate(*trn_passives*):**
        a. load *passive* into passive volume
        b. **for muon_batch in range(*n_mu_per_volume*//*mu_bs*):**
            i. Irradiate volume with *mu_bs* muons
            ii. Infer passive volume
        c. Compute loss based on precision and cost, and add to *loss*
        d. **if *p* `+1 % `passive_bs == 0:**
            i. *loss* = *loss*/*passive_bs*
            ii. Backpropagate *loss* and update detector parameters
            iii. *loss* = 0
        e. **if len(*trn_passives*)-(*p* `+1) < `passive_bs:**
            i. Break
    C. *val_loss* = 0
    D. **for *p*, *passive* in enumerate(*val_passives*):**
        a. load *passive* into passive volume
        b. **for muon_batch in range(*n_mu_per_volume*//*mu_bs*):**
            i. Irradiate volume with *mu_bs* muons
            ii. Infer passive volume
            iii. Compute loss based on precision and cost, and add to *val_loss*

c. **if len(*val_passives*)-(*p `+1) < `passive_bs*:**

i. Break

E. *val_loss* = *val_loss*/*p*

In implementation, the loop is broken up into several functions:

`_fit_epoch()` runs one full epoch of volumes
and updates for both training and validation

`_scan_volumes()` runs over all training/validation volumes,
updating parameters when necessary

`_scan_volume()` irradiates a single volume with muons multiple batches,
and computes the loss for that volume

The optimisation and prediction loops are supported by a stateful callback mechanism. The base callback is `Callback`, which can interject at various points in the loops. All aspects of the optimisation and prediction are stored in a `FitParams` data class, since the callbacks are also stored there, and the callbacks have a reference to the wrapper, they are able to read/write to the *FitParams* and be aware of other callbacks that are running.

Accounting for the interjection calls (*on_*_begin* & *on_*_end*), the full optimisation loop is:

1. Associate callbacks with wrapper (*set_wrapper*)
2. *on_train_begin*
3. **for epoch in *n_epochs*:**

A. *state* = "train"

B. *on_epoch_begin*

C. **for *p*, *passive* in enumerate(*trn_passives*):**

a. **if *p* % *passive_bs* == 0:**

i. *on_volume_batch_begin*

ii. *loss* = 0

b. load *passive* into passive volume

c. *on_volume_begin*

d. **for muon_batch in range(*n_mu_per_volume*//*mu_bs*):**

i. *on_mu_batch_begin*

ii. Irradiate volume with *mu_bs* muons

iii. Infer scatter locations

iv. *on_scatter_end*

v. Infer x0 and append to list of x0 predictions

vi. *on_mu_batch_end*

e. *on_x0_pred_begin*

f. Compute overall x0 prediction

g. *on_x0_pred_end*

h. Compute loss based on precision and cost, and add to *loss*

i. **if *p `+1* % *`passive_bs* == 0:**

i. *loss* = *loss*/*passive_bs*

ii. *on_volume_batch_end*

iii. Zero parameter gradients

iv. *on_backwards_begin*

v. Backpropagate *loss* and compute parameter gradients

vi. *on_backwards_end*

vii. Update detector parameters

viii. Ensure detector parameters are within physical boundaries (*AbsDetectorLayer.conform_detector*) viv. *loss* = 0

j. **if len(*trn_passives*)-(*p `+1) < `passive_bs*:**

i. Break

D. *on_epoch_end*

E. *state* = "valid"

F. *on_epoch_begin*

G. **for *p*, *passive* in enumerate(*val_passives*):**

a. **if *p %o passive_bs* == 0:**

i. *on_volume_batch_begin*

ii. *loss* = 0

b. *on_volume_begin*

c. **for muon_batch in range(*n_mu_per_volume//mu_bs*):**

i. *on_mu_batch_begin*

ii. Irradiate volume with *mu_bs* muons

iii. Infer scatter locations

iv. *on_scatter_end*

v. Infer x0 and append to list of x0 predictions

vi. *on_mu_batch_end*

d. *on_x0_pred_begin*

e. Compute overall x0 prediction

f. *on_x0_pred_end*

g. Compute loss based on precision and cost, and add to *loss*

h. **if *p `+1 % `passive_bs* == 0:**

i. *loss = loss/passive_bs*

ii. *on_volume_batch_end*

i. **if len(*val_passives*)-(*p `+1) < `passive_bs*:**

i. Break

H. *on_epoch_end*

4. *on_train_end*

Parameters

- **volume** (*Volume*) – the volume containing the detectors to be optimised
- **mu_opt** (Callable[[Iterator[Parameter]], Optimizer]) – uninitialised optimiser to be used for adjusting the xy position of Gaussians
- **norm_opt** (Callable[[Iterator[Parameter]], Optimizer]) – uninitialised optimiser to be used for adjusting the normalisation of Gaussians
- **sig_opt** (Callable[[Iterator[Parameter]], Optimizer]) – uninitialised optimiser to be used for adjusting the scale of Gaussians
- **z_pos_opt** (Callable[[Iterator[Parameter]], Optimizer]) – uninitialised optimiser to be used for adjusting the z position of panels
- **loss_func** (Optional[*AbsDetectorLoss*]) – optional loss function (required if planning to optimise the detectors)
- **partial_scatter_inferrer** (Type[*ScatterBatch*]) – uninitialised class to be used for inferring muon scatter variables and trajectories
- **partial_volume_inferrer** (Type[*AbsVolumeInferrer*]) – uninitialised class to be used for inferring volume targets
- **mu_generator** (Optional[*AbsMuonGenerator*]) – Optional generator class for muons. If None, will use from_volume().

classmethod from_save(*name*, ***, *volume*, *mu_opt*, *norm_opt*, *sig_opt*, *z_pos_opt*, *loss_func*, *partial_scatter_inferrer=<class 'tomopt.inference.scattering.ScatterBatch'>*, *partial_volume_inferrer=<class 'tomopt.inference.volume.PanelX0Inferrer'>*, *mu_generator=None*)

Instantiates a new *HeatMapVolumeWrapper* and loads saved detector and optimiser parameters

Parameters

- **name** (str) – file name with saved detector and optimiser parameters
- **volume** (*Volume*) – the volume containing the detectors to be optimised
- **mu_opt** (Callable[[Iterator[Parameter]], Optimizer]) – uninitialised optimiser to be used for adjusting the xy position of Gaussians
- **norm_opt** (Callable[[Iterator[Parameter]], Optimizer]) – uninitialised optimiser to be used for adjusting the normalisation of Gaussians
- **sig_opt** (Callable[[Iterator[Parameter]], Optimizer]) – uninitialised optimiser to be used for adjusting the scale of Gaussians
- **z_pos_opt** (Callable[[Iterator[Parameter]], Optimizer]) – uninitialised optimiser to be used for adjusting the z position of panels
- **loss_func** (Optional[*AbsDetectorLoss*]) – optional loss function (required if planning to optimise the detectors)
- **partial_scatter_inferrer** (Type[*ScatterBatch*]) – uninitialised class to be used for inferring muon scatter variables and trajectories
- **partial_volume_inferrer** (Type[*AbsVolumeInferrer*]) – uninitialised class to be used for inferring volume targets
- **mu_generator** (Optional[*AbsMuonGenerator*]) – Optional generator class for muons. If None, will use from_volume().

Return type

AbsVolumeWrapper

class tomopt.optimisation.wrapper.volume_wrapper.**PanelVolumeWrapper**(*volume*, *, *xy_pos_opt*, *z_pos_opt*, *xy_span_opt*, *budget_opt=None*, *loss_func=None*, *partial_scatter_inferrer=<class 'tomopt.inference.scattering.ScatterBatch'>*, *partial_volume_inferrer=<class 'tomopt.inference.volume.PanelX0Inferrer'>*, *mu_generator=None*)

Bases: *AbsVolumeWrapper*

Volume wrapper for volumes with *DetectorPanel*-based detectors.

Volume wrappers are designed to contain a *Volume* and provide means of optimising the detectors it contains, via their *fit()* method.

Wrappers also provide for various quality-of-life methods, such as saving and loading detector configurations, and computing predictions with a fixed detector (*predict()*)

Fitting of a detector proceeds as training and validation epochs, each of which contains multiple batches of passive volumes. For each volume in a batch, the loss is evaluated using multiple batches of muons. The whole loop is:

1. **for epoch in *n_epochs*:**
    A. *loss* = 0
    B. **for *p*, *passive* in enumerate(*trn_passives*):**
        a. load *passive* into passive volume
        b. **for muon_batch in range(*n_mu_per_volume*//*mu_bs*):**
            i. Irradiate volume with *mu_bs* muons
            ii. Infer passive volume
        c. Compute loss based on precision and cost, and add to *loss*
        d. **if *p* `+1 % `passive_bs == 0:**
            i. *loss* = *loss*/*passive_bs*
            ii. Backpropagate *loss* and update detector parameters
            iii. *loss* = 0
        e. **if len(*trn_passives*)-(*p* `+1) < `passive_bs:**
            i. Break
    C. *val_loss* = 0
    D. **for *p*, *passive* in enumerate(*val_passives*):**
        a. load *passive* into passive volume
        b. **for muon_batch in range(*n_mu_per_volume*//*mu_bs*):**
            i. Irradiate volume with *mu_bs* muons
            ii. Infer passive volume
            iii. Compute loss based on precision and cost, and add to *val_loss*

c. **if len(*val_passives*)-(*p `+1) < `passive_bs*:**

i. Break

E. *val_loss* = *val_loss*/*p*

In implementation, the loop is broken up into several functions:

`_fit_epoch()` runs one full epoch of volumes
and updates for both training and validation

`_scan_volumes()` runs over all training/validation volumes,
updating parameters when necessary

`_scan_volume()` irradiates a single volume with muons multiple batches,
and computes the loss for that volume

The optimisation and prediction loops are supported by a stateful callback mechanism. The base callback is `Callback`, which can interject at various points in the loops. All aspects of the optimisation and prediction are stored in a `FitParams` data class, since the callbacks are also stored there, and the callbacks have a reference to the wrapper, they are able to read/write to the *FitParams* and be aware of other callbacks that are running.

Accounting for the interjection calls (*on_*_begin* & *on_*_end*), the full optimisation loop is:

1. Associate callbacks with wrapper (*set_wrapper*)
2. *on_train_begin*
3. **for epoch in *n_epochs*:**

A. *state* = "train"

B. *on_epoch_begin*

C. **for *p*, *passive* in enumerate(*trn_passives*):**

a. **if *p* % *passive_bs* == 0:**

i. *on_volume_batch_begin*

ii. *loss* = 0

b. load *passive* into passive volume

c. *on_volume_begin*

d. **for muon_batch in range(*n_mu_per_volume*//*mu_bs*):**

i. *on_mu_batch_begin*

ii. Irradiate volume with *mu_bs* muons

iii. Infer scatter locations

iv. *on_scatter_end*

v. Infer x0 and append to list of x0 predictions

vi. *on_mu_batch_end*

e. *on_x0_pred_begin*

f. Compute overall x0 prediction

g. *on_x0_pred_end*

h. Compute loss based on precision and cost, and add to *loss*

i. **if *p `+1* % *`passive_bs* == 0:**

i. *loss* = *loss*/*passive_bs*

ii. *on_volume_batch_end*

iii. Zero parameter gradients

iv. *on_backwards_begin*

v. Backpropagate *loss* and compute parameter gradients

vi. *on_backwards_end*

vii. Update detector parameters

viii. Ensure detector parameters are within physical boundaries (*AbsDetectorLayer.conform_detector*) viv. *loss* = 0

j. **if len(*trn_passives*)-(*p `+1) < `passive_bs*:**

i. Break

D. *on_epoch_end*

E. *state* = "valid"

F. *on_epoch_begin*

G. **for *p*, *passive* in enumerate(*val_passives*):**

a. **if *p %o passive_bs* == 0:**

i. *on_volume_batch_begin*

ii. *loss* = 0

b. *on_volume_begin*

c. **for muon_batch in range(*n_mu_per_volume//mu_bs*):**

i. *on_mu_batch_begin*

ii. Irradiate volume with *mu_bs* muons

iii. Infer scatter locations

iv. *on_scatter_end*

v. Infer x0 and append to list of x0 predictions

vi. *on_mu_batch_end*

d. *on_x0_pred_begin*

e. Compute overall x0 prediction

f. *on_x0_pred_end*

g. Compute loss based on precision and cost, and add to *loss*

h. **if *p `+1 % `passive_bs* == 0:**

i. *loss = loss/passive_bs*

ii. *on_volume_batch_end*

i. **if len(*val_passives*)-(*p `+1) < `passive_bs*:**

i. Break

H. *on_epoch_end*

4. *on_train_end*

Parameters

- **volume** (*Volume*) – the volume containing the detectors to be optimised
- **xy_pos_opt** (Callable[[Iterator[Parameter]], Optimizer]) – uninitialised optimiser to be used for adjusting the xy position of panels
- **z_pos_opt** (Callable[[Iterator[Parameter]], Optimizer]) – uninitialised optimiser to be used for adjusting the z position of panels
- **xy_span_opt** (Callable[[Iterator[Parameter]], Optimizer]) – uninitialised optimiser to be used for adjusting the xy size of panels
- **budget_opt** (Optional[Callable[[Iterator[Parameter]], Optimizer]]) – optional uninitialised optimiser to be used for adjusting the fractional assignment of budget to the panels
- **loss_func** (Optional[*AbsDetectorLoss*]) – optional loss function (required if planning to optimise the detectors)
- **partial_scatter_inferrer** (Type[*ScatterBatch*]) – uninitialised class to be used for inferring muon scatter variables and trajectories
- **partial_volume_inferrer** (Type[*AbsVolumeInferrer*]) – uninitialised class to be used for inferring volume targets
- **mu_generator** (Optional[*AbsMuonGenerator*]) – Optional generator class for muons. If None, will use `from_volume()`.

classmethod from_save(*name*, ***, *volume*, *xy_pos_opt*, *z_pos_opt*, *xy_span_opt*, *budget_opt=None*, *loss_func*, *partial_scatter_inferrer=<class 'tomopt.inference.scattering.ScatterBatch'>*, *partial_volume_inferrer=<class 'tomopt.inference.volume.PanelX0Inferrer'>*, *mu_generator=None*)

Instantiates a new *PanelVolumeWrapper* and loads saved detector and optimiser parameters

Parameters

- **name** (str) – file name with saved detector and optimiser parameters
- **volume** (*Volume*) – the volume containing the detectors to be optimised
- **xy_pos_opt** (Callable[[Iterator[Parameter]], Optimizer]) – uninitialised optimiser to be used for adjusting the xy position of panels
- **z_pos_opt** (Callable[[Iterator[Parameter]], Optimizer]) – uninitialised optimiser to be used for adjusting the z position of panels,
- **xy_span_opt** (Callable[[Iterator[Parameter]], Optimizer]) – uninitialised optimiser to be used for adjusting the xy size of panels,
- **budget_opt** (Optional[Callable[[Iterator[Parameter]], Optimizer]]) – optional uninitialised optimiser to be used for adjusting the fractional assignment of budget to the panels
- **loss_func** (Optional[*AbsDetectorLoss*]) – optional loss function (required if planning to optimise the detectors)
- **partial_scatter_inferrer** (Type[*ScatterBatch*]) – uninitialised class to be used for inferring muon scatter variables and trajectories
- **partial_volume_inferrer** (Type[*AbsVolumeInferrer*]) – uninitialised class to be used for inferring volume targets
- **mu_generator** (Optional[*AbsMuonGenerator*]) – Optional generator class for muons. If None, will use `from_volume()`.

Return type
AbsVolumeWrapper

tomopt.plotting package

Submodules

tomopt.plotting.appearance module

tomopt.plotting.diagnostics module

tomopt.plotting.diagnostics.**plot_hit_density**(*hit_df*, *savename=None*)

Plots the position of muon hits in the detectors, as recorded using *HitRecord*.

Parameters

- **hit_df** (DataFrame) – Dataframe of recorded hits, as returned by `get_record()`
- **savename** (Optional[str]) – optional savename to save the plot

Return type
None

tomopt.plotting.diagnostics.**plot_scatter_density**(*scatter_df*, *savename=None*)

Plots the position of PoCAs in the passive volume, as recorded using *ScatterRecord*.

Parameters

- **scatter_df** (DataFrame) – Dataframe of recorded PoCAs, as returned by *get_record()*
- **savename** (Optional[str]) – optional savename to save the plot

Return type
None

tomopt.plotting.predictions module

tomopt.plotting.predictions.**plot_pred_true_x0**(*pred*, *true*, *savename=None*)

Plots the predicted voxelwise X0s compared to the true values of the X0s. 2D plots are produced in xy for layers in z in order of increasing z, i.e. the bottom most layer is the first to be plotted. TODO: revise this ordering to make it more intuitive

Parameters

- **pred** (ndarray) – (z,x,y) array of predicted X0s
- **true** (ndarray) – (z,x,y) array of true X0s
- **savename** (Optional[str]) – optional savename for saving the plot

Return type
None

tomopt.volume package

Submodules

tomopt.volume.heatmap module

class tomopt.volume.heatmap.**DetectorHeatMap**(*, *res*, *eff*, *init_xyz*, *init_xy_span*, *m2_cost=1*, *budget=None*, *realistic_validation=False*, *device=device(type='cpu')*, *n_cluster=30*)

Bases: Module

assign_budget(*budget=None*)

Return type
None

clamp_params(*musigz_low*, *musigz_high*)

Return type
None

get_cost()

Return type
Tensor

get_efficiency(*xy*, *mask=None*, *as_2d=False*)

Return type
Tensor

get_hits(*mu*)

Return type
Dict[str, Tensor]

get_resolution(*xy*, *mask=None*)

Return type
Tensor

get_xy_mask(*xy*)

Return type
Tensor

plot_map(*bpixelate=False*, *bsavefig=False*, *filename=None*)

Return type
None

property x: Tensor

property y: Tensor

tomopt.volume.layer module

class tomopt.volume.layer.**AbsDetectorLayer**(*pos*, ***, *lw*, *z*, *size*, *device=device(type='cpu')*)

Bases: *AbsLayer*

Abstract base class for layers designed to record muon positions (hits) using detectors. Inheriting classes should override a number methods to do with costs/budgets, and hit recording.

When optimisation of operating in 'fixed budget' mode, the *Volume* will check the *_n_costs* class attribute of the layer and will add this to the total number of learnable budget assignments, and pass that number of budgets as an (_n_costs) tensor. By default this is zero, and inheriting classes should set the correct number during initialisation, or via a new default value.

Some parts of TomOpt act differently on detector layers, according to how the detectors are modelled. A *type_label* attribute is used to encode extra information, rather than relying purely on the object-instance type.

Multiple detection layers can be grouped together, via their *pos* attribute (position); a string-encoded value. By default, the inference methods expect detectors above the passive layer to have *pos*=='*above*', and those below the passive volume to have *pos*=='*below*'. When retrieving hits from the muon batch, hits will be stacked together with other hits from the same *pos*.

The length and width (*lw*) is the spans of the layer in metres in x and y, and the layer begins at x=0, y=0. z indicates the position of the top of the layer, in meters, and size is the distance from the top of the layer to the bottom.

Important: By default, the detectors will not scatter muons.

Parameters

- **pos** (str) – string-encoding of the detector-layer group
- **lw** (Tensor) – the length and width of the layer in the x and y axes in metres, starting from (x,y)=(0,0).
- **z** (float) – the z position of the top of layer in metres. The bottom of the layer will be located at z-size
- **size** (float) – the voxel size in metres. Must be such that lw is divisible by the specified size.
- **device** (device) – device on which to place tensors

assign_budget(*budget*)

Inheriting classes should override this method to correctly assign elements of an (_n_costs) tensor to the parts of the detector to which they relate. All ordering of the tensor is defined using the function, but proper optimisation of the budgets may require that the same ordering is used, or that it is deterministic.

Parameters

budget (Optional[Tensor]) – (_n_costs) tensor of budget assignments in unit currency

Return type

None

conform_detector()

Optional method designed to ensure that the detector parameters lie within any require boundaries, etc. It will be called via the AbsVolumeWrapper after any update to the detector layers, but by default does nothing.

Return type
None

abstract forward(*mu*)

Inheriting classes should override this method to implement the passage of the muons through the layer, and record muon positions (hits) according to the detector model.

Parameters
mu (*MuonBatch*) – the incoming batch of muons

Return type
None

abstract get_cost()

Inheriting classes should override this method to return the total, current cost of the detector(s) in the layer.

Return type
Tensor

Returns
Single-element tensor with the current total cost of the detector in the layer.

class tomopt.volume.layer.**AbsLayer**(*lw*, *z*, *size*, *device=device(type='cpu')*)

Bases: Module

Abstract base class for volume layers. The length and width (*lw*) is the spans of the layer in metres in x and y, and the layer begins at x=0, y=0. z indicates the position of the top of the layer, in meters, and size is the distance from the top of the layer to the bottom. size is also used to set the length, width, and height of the voxels that make up the layer.

Important: Users must ensure that both the length and width of the layer are divisible by size

---

Parameters

- **lw** (Tensor) – the length and width of the layer in the x and y axes in metres, starting from (x,y)=(0,0).
- **z** (float) – the z position of the top of layer in metres. The bottom of the layer will be located at z-size
- **size** (float) – the voxel size in metres. Must be such that lw is divisible by the specified size.
- **device** (device) – device on which to place tensors

abstract forward(*mu*)

Inheriting classes should override this method to implement the passage of the muons through the layer.

Parameters
mu (*MuonBatch*) – the incoming batch of muons

Return type
None

get_lw_z_size()

Return type
Tuple[Tensor, Tensor, float]

Returns
The length and width of the layer in the x and y axes in metres, starting from (x,y)=(0,0), the z position of the top of layer in metres, and the voxel size in metres.

class tomopt.volume.layer.**PanelDetectorLayer**(*pos*, *, *lw*, *z*, *size*, *panels*)

Bases: *AbsDetectorLayer*

A detector layer class that uses multiple "panels" to record muon positions (hits). Currently, two "panel" types are available: *DetectorPanel* and *DetectorHeatMap* Each detector layer, however, should contain the same type of panel, as this is used to set the *type_label* of the layer.

When optimisation of operating in 'fixed budget' mode, the *Volume* will check the *_n_costs* class attribute of the layer and will add this to the total number of learnable budget assignments, and pass that number of budgets as an (_n_costs) tensor. During initialisation, this is set to the number of panels in the layer, at time of initialisation.

Multiple detection layers can be grouped together, via their *pos* attribute (position); a string-encoded value. By default, the inference methods expect detectors above the passive layer to have *pos*=='*above*', and those below the passive volume to have *pos*=='*below*'. When retrieving hits from the muon batch, hits will be stacked together with other hits from the same *pos*.

The length and width (*lw*) is the spans of the layer in metres in x and y, and the layer begins at x=0, y=0. z indicates the position of the top of the layer, in meters, and size is the distance from the top of the layer to the bottom.

Important: The detector panels do not scatter muons.

Parameters

- **pos** (str) – string-encoding of the detector-layer group
- **lw** (Tensor) – the length and width of the layer in the x and y axes in metres, starting from (x,y)=(0,0).
- **z** (float) – the z position of the top of layer in metres. The bottom of the layer will be located at z-size
- **size** (float) – the voxel size in metres. Must be such that lw is divisible by the specified size.
- **panels** (Union[List[*DetectorPanel*], List[*DetectorHeatMap*], List[*SigmoidDetectorPanel*], ModuleList]) – The set of initialised panels to contain in the detector layer

assign_budget(*budget*)

Passes elements of an (_n_costs) tensor to each of the panels' *assign_budget* method. Panels are ordered by decreasing z-position, i.e. the zeroth budget element will relate always to the highest panel, rather than necessarily to the same panel through the optimisation process

TODO investigate whether it would be better to instead assign budgets based on a fixed ordering, rather than the z-order of the panels.

Parameters
budget (Optional[Tensor]) – (_n_costs) tensor of budget assignments in unit currency

Return type
None

conform_detector()

Loops through panels and calls their *clamp_params* method, to ensure that panels are located within the bounds of the detector layer. It will be called via the `AbsVolumeWrapper` after any update to the detector layers.

Return type
None

forward(*mu*)

Propagates muons to each detector panel, in order of decreasing z-position, and calls their *get_hits* method to record hits to the muon batch. After this, the muons will be propagated to the bottom of the detector layer.

Parameters
mu (*MuonBatch*) – the incoming batch of muons

Return type
None

get_cost()

Returns the total, current cost of the detector(s) in the layer, as computed by looping over the panels and summing the returned values of calls to their *get_cost* methods.

Return type
Tensor

Returns
Single-element tensor with the current total cost of the detector in the layer.

static get_device(*panels*)

Helper method to ensure that all panels are on the same device, and return that device. If not all the panels are on the same device, then an exception will be raised.

Parameters
panels (ModuleList) – ModuleLists of either *DetectorPanel* or *DetectorHeatMap* objects on device

Return type
device

Returns
Device on which all the panels are.

get_panel_zorder()

Return type
List[int]

Returns
The indices of the panels in order of decreasing z-position.

yield_zordered_panels()

Yields the index of the panel, and the panel, in order of decreasing z-position.

Return type
Union[Iterator[Tuple[int, *DetectorPanel*]], Iterator[Tuple[int, *DetectorHeatMap*]]]

Returns
Iterator yielding panel indices and panels in order of decreasing z-position.

class tomopt.volume.layer.**PassiveLayer**(*lw*, *z*, *size*, *rad_length_func=None*, *step_sz=0.01*, *scatter_model='pdg'*, *device=device(type='cpu')*)

Bases: *AbsLayer*

Default layer of containing passive material that scatters the muons. The length and width (*lw*) is the spans of the layer in metres in x and y, and the layer begins at x=0, y=0. z indicates the position of the top of the layer, in meters, and size is the distance from the top of the layer to the bottom. size is also used to set the length, width, and height of the voxels that make up the layer.

Important: Users must ensure that both the length and width of the layer are divisible by size

If the layer is set to scatter muons (*rad_length* is not None), then two scattering models are available:

- 'pdg': The default and currently recommended model based on the Gaussian scattering model described in https://pdg.lbl.gov/2019/reviews/rpp2018-rev-passage-particles-matter.pdf
- 'pgeant': An under-development model based on a parameterised fit to data sampled from GEANT 4

The X0 values of each voxel is defined via a "radiation-length function", which should return an (n_x,n_y) tensor of voxel X0 values, when called with the *z*, *lw*, and *size* of the layer. For example:

```
def arb_rad_length(*, z: float, lw: Tensor, size: float) -> float:
    rad_length = torch.ones(list((lw / size).long())) * X0["lead"]
    if z < 0.5:
        rad_length[...] = X0["beryllium"]
    return rad_length
```

This function can either be supplied during initialisation, or later via the *load_rad_length* method.

Parameters

- **lw** (Tensor) – the length and width of the layer in the x and y axes in metres, starting from (x,y)=(0,0).
- **z** (float) – the z position of the top of layer in metres. The bottom of the layer will be located at z-size
- **size** (float) – the voxel size in metres. Must be such that lw is divisible by the specified size.
- **rad_length_func** (Optional[Callable[[Tensor, Tensor, float], Tensor]]) – lookup function that returns an (n_x,n_y) tensor of voxel X0 values for the layer. After initialisation, the *load_rad_length* method may be used to load X0 layouts.
- **step_sz** (float) – The step size in metres over which to compute muon propagation and scattering.
- **scatter_model** (str) – String selection for the scattering model to use. Currently either 'pdg' or 'pgeant'.
- **device** (device) – device on which to place tensors

abs2idx(*xy*)

Helper method to return the voxel indices in the layer of the supplied tensor of xy positions.

Warning: This method does NOT account for the possibility of positions may be outside the layer. Please ensure that positions are inside the layer.

Parameters
xy (Tensor) – (N,xy) tensor of absolute xy positions in metres in the volume frame

Return type
Tensor

Returns
(N,xy) tensor of voxel indices in x,y

forward(*mu*)

Propagates the muons through the layer to the bottom in a series of scattering steps. If the 'pdg' model is used, then the step size is the *step_sz* of the layer, as supplied during initialisation. If the 'pgeant' model is used, the the step size specified as part of the fitting of the scattering model.

Parameters
mu (*MuonBatch*) – the incoming batch of muons

Return type
None

load_rad_length(*rad_length_func*)

Loads a new X0 layout into the layer voxels.

Parameters
rad_length_func (Callable[[Tensor, Tensor, float], Tensor]) – lookup function that returns an (n_x,n_y) tensor of voxel X0 values for the layer.

Return type
None

mu_abs2idx(*mu*, *mask=None*)

Helper method to return the voxel indices in the layer that muons currently occupy.

> **Warning:** This method does NOT account for the possibility of muons being outside the layer. Please also supply a mask, to only select muons inside the layer.

Parameters

- **mu** (*MuonBatch*) – muons to look up
- **mask** (Optional[Tensor]) – Optional (muons) Boolean tensor where True indicates that the muon position should be checked

Return type
Tensor

Returns
(muons,2) tensor of voxel indices in x,y

scatter_and_propagate(*mu*, *mask=None*)

Propagates the muons through (part of) the layer by the prespecified *step_sz*. If the layer is set to scatter muons (*rad_length* is not None), then the muons will also undergo scattering (changes in their trajectories and positions) according to the scatter model of the layer.

Warning: When computing scatterings, the X0 used for each muon is that of the starting voxel: If a muon moves into a neighbouring voxel of differing X0, then this will only be accounted for in the next step.

Parameters

- **mu** (*MuonBatch*) – muons to propagate
- **mask** (Optional[Tensor]) – Optional (N,) Boolean mask. Only muons with True values will be scattered and propagated

Return type
None

tomopt.volume.panel module

class tomopt.volume.panel.**DetectorPanel**(*, *res*, *eff*, *init_xyz*, *init_xy_span*, *m2_cost=1*, *budget=None*, *realistic_validation=True*, *device=device(type='cpu')*)

Bases: Module

Provides an infinitely thin, rectangular panel in the xy plane, centred at a learnable xyz position (metres, in absolute position in the volume frame), with a learnable width in x and y (*xy_span*). Whilst this class can be used manually, it is designed to be used by the *PanelDetectorLayer* class.

Despite inheriting from *nn.Module*, the *forward* method should not be called, instead passing a *MuonBatch* to the *get_hits* method will return hits corresponding to the muons.

During training model (*.train()* or *.training* is True), a continuous model of the resolution and efficiency will be used, such that hits are differentiable w.r.t. the learnable parameters of the panel. This means that muons outside of the physical panel will have hits at non-zero resolution. The current model is a 2D uncorrelated Gaussian in xy, centred over the panel, with width parameters equal to the xy_spans/4, i.e. the panel is 4-sigma across.

Efficiency is accounted for via a weighting approach, rather than deciding whether to record hits or not, i.e. muons will always record hits, but the probability of the hit actually being recorded is computable.

During evaluation mode (*.eval()* or *.training* is False), if the panel is set to use *realistic_validation*, then the physical panel will be simulated: Muons outside the panel have zero resolution and efficiency, resulting in NaN hits positions. Muons inside the panel will have the full resolution and efficiency of the panel, but hits will not be differentiable w.r.t. the panel xy-position or xy-span. If *realistic_validation* is False, then the continuous model will be used also in evaluation mode.

The cost of the panel is based on the supplied cost per metre squared, and the current area of the panel, according to its learnt xy-span. Panels can also be run in "fixed-budget" mode, in which a cost of the panel is specified via the *.assign_budget* method. Based on the cost per m^2, the panel will change in effective width, based on its learn xy_span (now an aspect ratio), such that its area results in a cost equal to the specified cost of the panel.

The resolution and efficiency remain fixed at the specified values. If intending to run in "fixed-budget" mode, then a budget can be specified here, however the :class"~*tomopt.volume.volume.Volume* class will pass through all panels and initialise their budgets.

Parameters

- **res** (float) – resolution of the panel in m^-1, i.e. a higher value improves the precision on the hit recording
- **eff** (float) – efficiency of the hit recording of the panel, indicated as a probability [0,1]

- **init_xyz** (Tuple[float, float, float]) – initial xyz position of the panel in metres in the volume frame
- **init_xy_span** (Tuple[float, float]) – initial xy-span (total width) of the panel in metres
- **m2_cost** (float) – the cost in unit currency of the 1 square metre of detector
- **budget** (Optional[Tensor]) – optional required cost of the panel. Based on the span and cost per m^2, the panel will resize to meet the required cost
- **realistic_validation** (bool) – if True, will use the physical interpretation of the panel during evaluation
- **device** (device) – device on which to place tensors

assign_budget(*budget=None*)

Sets the budget for the panel. This is then used to set a multiplicative coefficient, *budget_scale*, based on the *m2_cost* which rescales the *xy_span* such that the area of the resulting panel matches the assigned budget.

Parameters
budget (Optional[Tensor]) – required cost of the panel in unit currency

Return type
None

clamp_params(*xyz_low*, *xyz_high*)

Ensures that the panel is centred within the supplied xyz range, and that the span of the panel is between xyz_high/20 and xyz_high*10. A small random number < 1e-3 is added/subtracted to the min/max z position of the panel, to ensure it doesn't overlap with other panels.

Parameters

- **xyz_low** (Tuple[float, float, float]) – minimum x,y,z values for the panel centre in metres
- **xyz_high** (Tuple[float, float, float]) – maximum x,y,z values for the panel centre in metres

Return type
None

forward()

Define the computation performed at every call.

Should be overridden by all subclasses. :rtype: None

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

---

get_cost()

Return type
Tensor

Returns
current cost of the panel according to its area and m2_cost

`get_efficiency`(*xy*, *mask=None*)

Computes the efficiency of panel at the supplied list of xy points. If running in evaluation mode with *realistic_validation*, then these will be the full efficiency of the panel for points inside the panel (indicated by the mask), and zero outside. Otherwise, the Gaussian model will be used.

Parameters

- **xy** (`Tensor`) – (N,) or (N,xy) tensor of positions
- **mask** (`Optional[Tensor]`) – optional pre-computed (N,) Boolean mask, where True indicates that the xy point is inside the panel. Only used in evaluation mode and if *realistic_validation* is True. If required, but not supplied, than will be computed automatically.

Return type
`Tensor`

Returns
eff, a (N,)tensor of the efficiency at the xy points

`get_gauss`()

Return type
`Normal`

Returns
A Gaussian distribution, with 2 uncorrelated components corresponding to x and y, centred at the xy position of the panel, and sigma = panel span/4

`get_hits`(*mu*)

Return type
`Dict[str, Tensor]`

The main interaction method with the panel: returns hits for the supplied muons. Hits consist of:

reco_xy: (muons,xy) tensor of reconstructed xy positions of muons included simulated resolution gen_xy: (muons,xy) tensor of generator-level (true) xy positions of muons z: z position of the panel

If running in evaluation mode with *realistic_validation*, then these will be the full resolution of the panel for points inside the panel (indicated by the mask), and zero outside. Otherwise, the Gaussian model will be used.

`get_resolution`(*xy*, *mask=None*)

Computes the xy resolutions of panel at the supplied list of xy points. If running in evaluation mode with *realistic_validation*, then these will be the full resolution of the panel for points inside the panel (indicated by the mask), and zero outside. Otherwise, the Gaussian model will be used.

Parameters

- **xy** (`Tensor`) – (N,xy) tensor of positions
- **mask** (`Optional[Tensor]`) – optional pre-computed (N,) Boolean mask, where True indicates that the xy point is inside the panel. Only used in evaluation mode and if *realistic_validation* is True. If required, but not supplied, than will be computed automatically.

Return type
`Tensor`

Returns
res, a (N,xy) tensor of the resolution at the xy points

get_scaled_xy_span()

Computes the effective size of the panel by rescaling based on the xy-span, cost per m^2, and budget.

Return type
Tensor

Returns
Rescaled xy-span such that the panel has a cost equal to the specified budget

get_xy_mask(*xy*)

Computes which of the xy points lie inside the physical panel.

Parameters
xy (Tensor) – xy2) tensor of points

Return type
Tensor

Returns
(N,) Boolean mask, where True indicates the point lies inside the panel

property x: Tensor

property y: Tensor

class tomopt.volume.panel.**SigmoidDetectorPanel**(*, *smooth*, *res*, *eff*, *init_xyz*, *init_xy_span*, *m2_cost=1*, *budget=None*, *realistic_validation=True*, *device=device(type='cpu')*)

Bases: *DetectorPanel*

Provides an infinitely thin, rectangular panel in the xy plane, centred at a learnable xyz position (metres, in absolute position in the volume frame), with a learnable width in x and y (*xy_span*). Whilst this class can be used manually, it is designed to be used by the *PanelDetectorLayer* class.

Despite inheriting from *nn.Module*, the *forward* method should not be called, instead passing a *MuonBatch* to the *get_hits* method will return hits corresponding to the muons.

During training model (*.train()* or *.training* is True), a continuous model of the resolution and efficiency will be used, such that hits are differentiable w.r.t. the learnable parameters of the panel. This means that muons outside of the physical panel will have hits at non-zero resolution. The model is a 2D uncorrelated Sigmoid in xy, centred over the panel, which pass 0.5 at the xy_spans/2. The smoothness of the sigmoid affects the rate of change of resolution|efficiency near the edge of the physical border: A higher smooth value provides a slower change, with higher resolution|efficiency outside the physical panel, whereas a lower smooth value provides a sharper transition, with lower sensitivity to muons outside the panel (and therefore more strongly approximated a physical panel). The *SigmoidPanelSmoothnessSchedule* can be used to anneal this smoothness during optimisation.

Efficiency is accounted for via a weighting approach, rather than deciding whether to record hits or not, i.e. muons will always record hits, but the probability of the hit actually being recorded is computable.

During evaluation mode (*.eval()* or *.training* is False), if the panel is set to use *realistic_validation*, then the physical panel will be simulated: Muons outside the panel have zero resolution and efficiency, resulting in NaN hits positions. Muons inside the panel will have the full resolution and efficiency of the panel, but hits will not be differentiable w.r.t. the panel xy-position or xy-span. If *realistic_validation* is False, then the continuous model will be used also in evaluation mode.

The cost of the panel is based on the supplied cost per metre squared, and the current area of the panel, according to its learnt xy-span. Panels can also be run in "fixed-budget" mode, in which a cost of the panel is specified via the *.assign_budget* method. Based on the cost per m^2, the panel will change in effective width, based on its learn xy_span (now an aspect ratio), such that its area results in a cost equal to the specified cost of the panel.

The resolution and efficiency remain fixed at the specified values. If intending to run in "fixed-budget" mode, then a budget can be specified here, however the :class"~*tomopt.volume.volume.Volume* class will pass through all panels and initialise their budgets.

Parameters

- **smooth** (Union[float, Tensor]) – smoothness of the sigmoid: A higher smooth value provides a slower change, with higher resolution|efficiency outside the physical panel, whereas a lower smooth value provides a sharper transition, with lower sensitivity to muons outside the panel (and therefore more strongly approximated a physical panel).
- **res** (float) – resolution of the panel in m^-1, i.e. a higher value improves the precision on the hit recording
- **eff** (float) – efficiency of the hit recording of the panel, indicated as a probability [0,1]
- **init_xyz** (Tuple[float, float, float]) – initial xyz position of the panel in metres in the volume frame
- **init_xy_span** (Tuple[float, float]) – initial xy-span (total width) of the panel in metres
- **m2_cost** (float) – the cost in unit currency of the 1 square metre of detector
- **budget** (Optional[Tensor]) – optional required cost of the panel. Based on the span and cost per m^2, the panel will resize to meet the required cost
- **realistic_validation** (bool) – if True, will use the physical interpretation of the panel during evaluation
- **device** (device) – device on which to place tensors

get_efficiency(*xy*, *mask=None*)

Computes the efficiency of panel at the supplied list of xy points. If running in evaluation mode with *realistic_validation*, then these will be the full efficiency of the panel for points inside the panel (indicated by the mask), and zero outside. Otherwise, the Sigmoid model will be used.

Parameters

- **xy** (Tensor) – (N,) or (N,xy) tensor of positions
- **mask** (Optional[Tensor]) – optional pre-computed (N,) Boolean mask, where True indicates that the xy point is inside the panel. Only used in evaluation mode and if *realistic_validation* is True. If required, but not supplied, than will be computed automatically.

Return type
Tensor

Returns
eff, a (N,)tensor of the efficiency at the xy points

get_resolution(*xy*, *mask=None*)

Computes the xy resolutions of panel at the supplied list of xy points. If running in evaluation mode with *realistic_validation*, then these will be the full resolution of the panel for points inside the panel (indicated by the mask), and zero outside. Otherwise, the Sigmoid model will be used.

Parameters

- **xy** (Tensor) – (N,xy) tensor of positions
- **mask** (Optional[Tensor]) – optional pre-computed (N,) Boolean mask, where True indicates that the xy point is inside the panel. Only used in evaluation mode and

if *realistic_validation* is True. If required, but not supplied, than will be computed automatically.

Return type
Tensor

Returns
res, a (N,xy) tensor of the resolution at the xy points

`sig_model`(*xy*)

Models fractional resolution and efficiency from a sigmoid-based model to provide a smooth and differentiable model of a physical detector-panel.

Parameters
xy (Tensor) – (N,xy) tensor of positions

Return type
Tensor

Returns
Multiplicative coefficients for the nominal resolution or efficiency of the panel based on the xy position relative to the panel position and size

`property smooth: Tensor`

tomopt.volume.scatter_model module

tomopt.volume.volume module

`class tomopt.volume.volume.Volume`(*layers*, *budget=None*)

Bases: Module

The *Volume* class is used to contain both passive layers and detector layers. It is designed to act as an interface to them for the convenience of e.g. VolumeWrapper, and to allow new passive-volume layouts to be loaded.

When optimisation is acting in *fixed-budget* mode, the volume is also responsible for learning the optimal assignments of the budget to detector parts.

Volumes can also have a "target" value. This could be e.g. the class ID of the passive-volume configuration which is currently loaded. See e.g. VolumeClassLoss. The target can be set as part of the call to *load_rad_length()*

The volume is expected to have its low-left-front (zxy) corner located at (0,0,0) metres.

Important: Currently this class expects that all *PassiveLayer* s form a single contiguous block, i.e. it does not currently support sparse, or multiple, passive volumes.

Parameters

- **layers** (ModuleList) – *torch.nn.ModuleList* of instantiated *AbsLayer* s, ordered in decreasing z position.
- **budget** (Optional[float]) – optional budget of the detector in currency units. Supplying a value for the optional budget, here, will prepare the volume to learn budget assignments to the detectors, and configure the detectors for the budget.

assign_budget()

Distributed the total budget for the detector system amongst the various sub-detectors. When assigning budgets to layers, the budget weights are softmax-normalised to one, and multiplied by the total budget. Slices of these budgets are then passed to the layers, with the length of the slices being taken from *_n_layer_costs*.

Return type
None

build_xyz_edges()

Computes the xyz locations of low-left-front edges of voxels in the passive layers of the volume.

Return type
Tensor

property device: device

draw(*, *xlim*, *ylim*, *zlim*)

Draws the layers/panels pertaining to the volume. When using this in a jupyter notebook, use "%matplotlib notebook" to have an interactive plot that you can rotate.

Parameters

- **xlim** (Tuple[float, float]) – the x axis range for the three-dimensional plot.
- **ylim** (Tuple[float, float]) – the y axis range for the three-dimensional plot.
- **zlim** (Tuple[float, float]) – the z axis range for the three-dimensional plot.

Return type
None

forward(*mu*)

Propagates muons through each layer in turn. Prior to propagating muons, the *assign_budget()* method is called.

Parameters
mu (*MuonBatch*) – the incoming batch of muons

Return type
None

get_cost()

Return type
Tensor

Returns

The total, current cost of the layers in the volume,
or the assigned budget for the volume (these two values should be the same but, the actual cost won't be evaluated explicitly)

get_detectors()

Return type
List[*AbsDetectorLayer*]

Returns
A list of all *AbsDetectorLayer* s in the volume, in the order of *layers* (normally decreasing z position)

get_passive_z_range()

Return type
Tuple[Tensor, Tensor]

Returns
The z position of the bottom of the lowest passive layer, and the z position of the top of the highest passive layer.

get_passives()

Return type
List[*PassiveLayer*]

Returns
A list of all *PassiveLayer* s in the volume, in the order of *layers* (normally decreasing z position)

get_rad_cube()

Return type
Tensor

Returns
zxy tensor of the values stored in the voxels of the passive volume, with the lowest layer being found in the zeroth z index position.

property h: Tensor

Returns: The height of the volume (including both passive and detector layers), as computed from the z position of the zeroth layer.

load_rad_length(*rad_length_func*, *target=None*)

Loads a new passive-volume configuration. Optionally, a "target" for the configuration may also be supplied. This could be e.g. the class ID of the passive-volume configuration which is currently loaded. See e.g. VolumeClassLoss.

Parameters

- **rad_length_func** (Callable[[Tensor, Tensor, float], Tensor]) – lookup function that returns an (n_x,n_y) tensor of voxel X0 values for the layer.
- **target** (Optional[Tensor]) – optional target for the new layout

Return type
None

lookup_passive_xyz_coords(*xyz*)

Looks up the voxel indices of the supplied list of absolute positions in the volume frame

> **Warning:** Assumes the same size for all passive layers, and that they form a single contiguous block

Parameters
xyz (Tensor) – an (N,xyz) tensor of absolute positions in the volume frame

Return type
Tensor

Returns
an (N,xyz) tensor of zero-ordered voxel indices, which correspond to the supplied positions

property lw: Tensor

Returns: The length and width of the passive volume

property passive_size: float

Returns: The size of voxels in the passive volume

property target: Tensor | None

Returns: The "target" value of the volume. This could be e.g. the class ID of the passive-volume configuration which is currently loaded. See e.g. `VolumeClassLoss`. The target can be set as part of the call to *load_rad_length()*

property xyz_centres: Tensor

xyz locations of the centres of voxels in the passive layers of the volume.

property xyz_edges: Tensor

xyz locations of low-left-front edges of voxels in the passive layers of the volume.

3.1.2 Submodules

3.1.3 tomopt.core module

3.1.4 tomopt.utils module

tomopt.utils.**class_to_x0preds**(*array*, *id2x0*)

Converts array of classes to X0 predictions using the map defined in id2x0

Parameters

- **array** (ndarray) – array of integer class IDs
- **id2x0** (Dict[int, float]) – map of class IDs to X0 float values

Return type
ndarray

Returns
New array of X0 values

tomopt.utils.**jacobian**(*y*, *x*, *create_graph=False*, *allow_unused=True*)

Computes the Jacobian (dy/dx) of y with respect to variables x. x and y can have multiple elements. If y has multiple elements then computation is vectorised via vmap.

Parameters

- **y** (Tensor) – tensor to be differentiated
- **x** (Tensor) – dependent variables
- **create_graph** (bool) – If True, graph of the derivative will be constructed, allowing to compute higher order derivative products. Default: False.
- **allow_unused** (bool) – If False, specifying inputs that were not used when computing outputs (and therefore their grad is always

Return type
Tensor

Returns
dy/dx tensor of shape y.shape+x.shape

tomopt.utils.**x0_from_mixture**(*x0s*, *densities*, *weight_fracs=None*, *volume_fracs=None*)

Computes the X0 of a mixture of (non-chemically bonded) materials, Based on https://cds.cern.ch/record/1279627/files/PH-EP-Tech-Note-2010-013.pdf

Parameters

- **x0s** (Union[ndarray, List[float]]) – X0 values of the materials in the mixture in metres
- **densities** (Union[ndarray, List[float]]) – densities of the materials in the mixture kg/m^3
- **weight_fracs** (Union[ndarray, List[float], None]) – the relative amounts of each material by weight
- **volume_fracs** (Union[ndarray, List[float], None]) – the relative amounts of each material by volume

Returns

The X0 of the defined mixture in metres, "density": The density in kg/m^3 of the defined mixture}

Return type

{"X0"

tomopt.utils.**x0targs_to_classtargs**(*array*, *x02id*)

Converts array of float X0 targets to integer class IDs using the map defined in x02id.

Warning: To account for floating point precision, X0 values are mapped to the integer class IDs which are closest to X0 keys defined in the map. This means that the method cannot detect missing X0 values from x02id; X0s will always be mapped to a class ID, even if the material isn't defined in the map

Warning: The input array is modified in-place

Parameters

- **array** (ndarray) – array of X0 values
- **x02id** (Dict[float, int]) – map of X0 float values to class IDs

Return type

ndarray

Returns

Array of integer class IDs

3.1.5 tomopt.version module

CHAPTER

FOUR

INDEX

- genindex

PYTHON MODULE INDEX

t

INDEX

D

E

F

G

H

I

J

L

M

N

O

P

U

V

W

X

Y

Z