LUMIN

LUMIN Contributors

Nov 19, 2021

PACKAGE DOCUMENTATION

1 Installation 3

2 Core concepts 5

3 lumin.data_processing package 11

4 lumin.evaluation package 23

5 lumin.inference package 25

6 lumin.nn package 27

7 lumin.optimisation package 111

8 lumin.plotting package 121

9 lumin.utils package 133

10 Package Description 139

11 Index 145

Python Module Index 147

Index 149

Lumin Unifies Many Improvements for Networks

LUMIN is a deep-learning and data-analysis ecosystem for High-Energy Physics, and perhaps other scientific domains in the future. Similar to Keras and fastai it is a wrapper framework for a graph computation library (PyTorch), but includes many useful functions to handle domain-specific requirements and problems. It also intends to provide easy access to to state-of-the-art methods, but still be flexible enough for users to inherit from base classes and override methods to meet their own demands.

CHAPTER

ONE

INSTALLATION

Due to some strict version requirements on packages, it is recommended to install LUMIN in its own Python environment, e.g `conda create -n lumin python=3.6`

1.1 From PyPI

The main package can be installed via: `pip install lumin`

Full functionality requires two additional packages as described below.

1.2 From source

```
git clone git@github.com:GilesStrong/lumin.git
cd lumin
pip install .
```

Optionally, run pip install with `-e` flag for development installation.

1.2.1 Optional requirements

- sparse: enables loading on COO sparse-format tensors, install via e.g. `pip install sparse`

CHAPTER

TWO

CORE CONCEPTS

2.1 The fold file

The fold file is the core data-structure used throughout LUMIN. It is stored on disc as an HDF5 file. In the top level are several groups. The `meta_data` group stores various datasets containing information about the data, such as the names of features. The other top-level groups are the *folds*. These store subsamples of the full dataset and are designed to be read into memory individually, and provide several advantages, such as:

- Memory requirements are reduced
- Specific fold indices can be designated for training and others for validation, e.g. for k-fold cross-validation
- Some methods can compute averaged metrics over folds and produce uncertainties based on standard deviation

Each fold group contains several datasets:

- `targets` will be used to provide target data for training NNs
- `inputs` contains the input data in the following
- `weights`, if present, will be used to weight losses during training
- `matrix_inputs` can be used to store 2D matrix, or higher-order (sparse) tensor data

Additional datasets can be added, too, e.g. extra features that are necessary for interpreting results. Named predictions can also be saved to the fold file e.g. during `Model.predict`. Datasets can also be compressed to reduce size and loading time.

2.1.1 Creating fold files

`lumin.data_processing.file_proc` contains the recommended methods for constructing fold files from `pandas.DataFrame` objects with the main construction method being df2foldfile, although the methods it calls can be used directly for complex or large data.

2.1.2 Reading fold files

The main interface class is the FoldYielder. Its primary function is to load data from the fold files, however it can also act as hub for meta information and objects concerning the dataset, such as feature names and processing pipelines. Specific features can be marked as 'ignore' and will be filtered out when loading folds.

Calling `fy.get_fold(i)` or indexing an instance `fy[i]` will return a dictionary of inputs, targets, and weights for fold `i` via the `get_data` method. Flat inputs will be passed through `np.nan_to_num`. If matrix or tensor inputs are present then they will be processed into a tuple with the flat data ([flat inputs, dense tensor]).

`fy.get_df` can be used to construct a `pandas.DataFrame` from the data (either specific folds, or all folds together). The method has various arguments for controlling what columns should be included. By default only targets, weights, and predictions are included. Additional datasets can also be loaded via the `get_column` method.

Since during training and inference folds are loaded into memory one at a time, used once, and overwritten LUMIN can optionally apply data augmentation when loading folds. The inheriting class HEPAugFoldYielder provides an example of this, where particle collision events can be rotated and flipped.

2.2 Models

2.2.1 Model building

Contrary to other high-level interfaces, in LUMIN the user is expected to define **how** models, optimisers, and loss functions should be built, rather than build them themselves. The ModelBuilder class helps to capture these definitions, and once instantiated, can be used produce models on demand.

LUMIN models consist of three types of *blocks*:

1. The Head, which takes all inputs from the data and processes them if necessary.
    - The default head is CatEmbHead, which passes continuous inputs through an optional dropout layer, and categorical inputs through embedding matrices (see Guo & Berkhahn, 2016) and an optional dropout layer.
    - Matrix or tensor data can also be passed through appropriate head blocks, e.g. RNNs, CNNs, and GNNs.
    - Data containing both matrix/tensor data and flat data (continuous+categorical) can be passed through a MultiHead block, which in turn sends data through the appropriate head and concatenates the outputs.
    - The output of the head is a flat vector (batch, head width)
2. The Body is where the majority of the computation occurs (at least in the case of a flat FCNN). The default body is FullyConnected, consisting of multiple hidden layers.
    - MultiBlock can be used to split features across separate body blocks, e.g. for wide-deep networks.
    - The output of the body is also a flat vector (batch, body width)
3. The Tail is designed to alter the body width to match the target width, as well as apply any output activation function and output rescaling.
    - The default tail is ClassRegMulti, which can handle single- & multi-target regression, and binary, multi-label, and multi-class classification (it configures itself using the `objective` attribute of the `ModelBuilder`).

The `ModelBuilder` has arguments to change the blocks from their default values. Custom blocks should be passed as classes, rather than instantiated objects (i.e. use `partial` to configure their arguments). There are some arguments for blocks which will be set automatically by the *ModelBuilder*: For heads, these are *cont_feats*, *cat_embedder*, *lookup_init*, and *freeze*; for bodies *n_in*, *feat_map*, *lookup_init*, *lookup_act*, and *freeze*; and for tails *n_in*, *n_out*, *objective*, *lookup_init*, and *freeze*. *model_args* can also be used to set arguments via a dictionary, e.g. *{'head':{'depth':3}}*.

The `ModelBuilder` also returns an optimiser set to update the parameters of the model. This can be configured via `opt_args` and custom optimisers can be passed as classes to `'opt'`, e.g. `opt_args={'opt':AdamW, 'lr':3e-2}`. The loss function is controlled by the `loss` argument, and can either be left as auto and set via the `objective`, or explicitly set to a class by the user. Use of pretrained models can be achieved by setting the `pretrain_file` argument to a previously trained model, which will then be loaded when a new model is built.

2.2.2 Model wrapper

The models built by the `ModelBuilder` are `torch.nn.Module` objects, and so to provide high-level functionality, LUMIN wraps these objects with a Model class. This provides a range of methods for e.g. training, saving, loading, and predicting with DNNs. The `torch.nn.Module` is set as the `Model.model` attribute.

A similar high-level wrapper class exists for ensembles (Ensemble), in which the methods extend over a range of `Model` objects.

2.3 Model training

`Model.fit` will train `Model.model` using data provided via a `FoldYielder`. A specific fold index can be set to be used as validation data, and the rest will be used as training data (or the user can specify explicitly which fold indices to use for training). Callbacks can be used to augment the training, as described later on. Training is 'stateful', with a `Model.fit_params` object having various attributes such as the data, current state (training or validation), and callbacks. Since each callback has the model as an attribute, they can access all aspects of the training via the `fit_params`.

Training proceeds thusly:

1. For epoch in epochs:
    1. Training epoch begins
        1. Training-fold indices are shuffled
        2. For fold in training folds (referred to as a *sub-epoch*):
            1. Load fold data into a BatchYielder, a class that yields batches of input, target, and weight data
            2. For batch in batches:
            3. Pass inputs `x` through network to get predictions `y_pred`
            4. Compute loss based on `y_pred` and targets `y`
            5. Back-propagate loss through network
            6. Update network parameters using optimiser
    2. Validation epoch begins
        1. Load validation-fold data into a `BatchYielder`
            1. For batch in batches:
            2. Pass inputs `x` through network to get predictions `y_pred`
            3. Compute loss based on `y_pred` and targets `y`

2.3.1 Training method

Whilst `Model.fit` can be used by the user, there is still a lot of boilerplate code that must be written to support convenient training and monitoring of models, plus one of the distinguishing characteristics of LUMIN is that training many models should be as easy as training one model. To this end, the recommended training function is train_models. This function will train a specified number of models and save them to a specified directory. It doesn't return the trained models, but rather a dictionary of results containing information about the training, and the paths to the models. This can then be used to instantiate an Ensemble via the `from_results` class-method.

2.4 Callbacks

Just like in Keras and FastAI, callbacks are a powerful and flexible way to augment the general training loop outlined above, by offering series of fine-grained interjection points:

- on_train_begin: after all preparations are made and the first epoch is about to begin; allows callbacks to initialise and prepare for the training
- on_epoch_begin: when a new training or validation epoch is about to begin
- on_fold_begin: when a new training or validation fold is about to begin and after the batch yielder has been instantiated; allows callbacks to modify the entirety of the data for the fold via `fit_params.by`
- on_batch_begin: when a new batch or data is about to be processed and inputs, targets, and weights have been set to `fit_params.x`, fit_params.y, `and fit_params.w`; allows callbacks to modify the batch before it is passed through the network
- on_forwards_end: after the inputs have been passed through the network and the predictions `fit_params.y_pred` and the loss value `fit_params.loss_val` computed; allows callbacks to modify the loss before it is back-propagated (e.g. adversarial training), or to compute a new loss value and set `fit_params.loss_val` manually
- on_backwards_begin: after the optimiser gradients have been zeroed and before the loss value has been back-propagated
- on_backwards_end: after the loss value has been back-propagated but before the optimiser update has been made; allows callbacks to modify the parameter gradients
- on_batch_end: after the batch has been processed, the loss computed, and any parameter updates made
- on_fold_end: after a training or validation fold has finished
- on_epoch_end: after a training or validation epoch has finished
- on_train_end: after the training has finished; allows callbacks to clean up and compute final results

In addition to callbacks during training, LUMIN offers callbacks at prediction, which can interject at:

- on_pred_begin: After all preparations are made and the prediction data has been loaded into a `BatchYielder`
- on_batch_begin
- on_forwards_end
- on_batch_end
- on_pred_end: After predictions have been made for al the data

Callbacks passed to the `Model` prediction methods come in two varieties: normal callbacks can be passed to `cbs`; and a special *prediction callback* can be passed to `pred_cb`. The prediction callback is responsible for storing and processing model predictions, and then returning the via a `get_preds` method. The default prediction callback simply returns predictions in the same order they were generated, however users may wish to e.g. rescale or bin predictions for convenience. An example use for other callbacks during prediction would be e.g. for inference of parameterised training model ParameterisedPrediction, Baldi et al., 2016.

2.4.1 Callbacks in LUMIN

A range of common, or useful, callbacks are provided in LUMIN:

- Optimiser and Cyclic callbacks are designed to modify optimiser hyper-parameters during training, e.g. OneCycle Smith, 2018. Classes inheriting from AbsCyclicCallback can signal to other callbacks to only act when a cycle has finished (e.g. stop training after no improvement).

- Data callbacks modify aspects of the data, e.g. for label smoothing, resampling, and replacing/removing values and data.
- Loss callbacks adjust the loss values and gradients, or even manually compute losses themselves.
- Model callbacks are a special type of callback that trains alternative models and can be polled for loss values, have their performance tracked, and have their models saved instead of the main model, e.g. SWA Izmailov et al., 2018.
- Monitor callbacks keep track of performance during the training, and provide a realtime report of metrics. Additionally, they can be used to save models when performance improves and stop training after improvements cease.
- Prediction handler callbacks are responsible for storing and adjusting the network outputs when predicting on new data.

CHAPTER

THREE

LUMIN.DATA_PROCESSING PACKAGE

3.1 Submodules

3.2 lumin.data_processing.file_proc module

lumin.data_processing.file_proc.**save_to_grp**(*arr*, *grp*, *name*, *compression=None*)

Save Numpy array as a dataset in an h5py Group

Parameters

- **arr** (ndarray) – array to be saved
- **grp** (Group) – group in which to save arr
- **name** (str) – name of dataset to create
- **compression** (Optional[str]) – optional compression argument for h5py, e.g. 'lzf'

Return type None

lumin.data_processing.file_proc.**fold2foldfile**(*df*, *out_file*, *fold_idx*, *cont_feats*, *cat_feats*, *targ_feats*, *targ_type*, *misc_feats=None*, *wgt_feat=None*, *matrix_lookup=None*, *matrix_missing=None*, *matrix_shape=None*, *tensor_data=None*, *compression=None*)

Save fold of data into an h5py Group

Parameters

- **df** (DataFrame) – Dataframe from which to save data
- **out_file** (File) – h5py file to save data in
- **fold_idx** (int) – ID for the fold; used name h5py group according to 'fold_{fold_idx}'
- **cont_feats** (List[str]) – list of columns in df to save as continuous variables
- **cat_feats** (List[str]) – list of columns in df to save as discreet variables
- **targ_feats** (Union[str, List[str]]) – (list of) column(s) in df to save as target feature(s)
- **targ_type** (Any) – type of target feature, e.g. int,'float32'
- **misc_feats** (Optional[List[str]]) – any extra columns to save
- **wgt_feat** (Optional[str]) – column to save as data weights

- **matrix_vecs** – list of objects for matrix encoding, i.e. feature prefixes
- **matrix_feats_per_vec** – list of features per vector for matrix encoding, i.e. feature suffixes. Features listed but not present in df will be replaced with NaN.
- **matrix_row_wise** – whether objects encoded as a matrix should be encoded row-wise (i.e. all the features associated with an object are in their own row), or column-wise (i.e. all the features associated with an object are in their own column)
- **tensor_data** (Optional[ndarray]) – data of higher order than a matrix can be passed directly as a numpy array, rather than beign extracted and reshaped from the DataFrame. The array will be saved under matrix data, and this is incompatible with also setting *matrix_lookup*, *matrix_missing*, and *matrix_shape*. The first dimension of the array must be compatible with the length of the data frame.
- **compression** (Optional[str]) – optional compression argument for h5py, e.g. 'lzf'

Return type None

lumin.data_processing.file_proc.**df2foldfile**(*df*, *n_folds*, *cont_feats*, *cat_feats*, *targ_feats*, *savename*, *targ_type*, *strat_key=None*, *misc_feats=None*, *wgt_feat=None*, *cat_maps=None*, *matrix_vecs=None*, *matrix_feats_per_vec=None*, *matrix_row_wise=None*, *tensor_data=None*, *tensor_name=None*, *tensor_is_sparse=False*, *compression=None*)

Convert dataframe into h5py file by splitting data into sub-folds to be accessed by a FoldYielder

Parameters

- **df** (DataFrame) – Dataframe from which to save data
- **n_folds** (int) – number of folds to split df into
- **cont_feats** (List[str]) – list of columns in df to save as continuous variables
- **cat_feats** (List[str]) – list of columns in df to save as discreet variables
- **targ_feats** (Union[str, List[str]]) – (list of) column(s) in df to save as target feature(s)
- **savename** (Union[Path, str]) – name of h5py file to create (.h5py extension not required)
- **targ_type** (str) – type of target feature, e.g. int,'float32'
- **strat_key** (Optional[str]) – column to use for stratified splitting
- **misc_feats** (Optional[List[str]]) – any extra columns to save
- **wgt_feat** (Optional[str]) – column to save as data weights
- **cat_maps** (Optional[Dict[str, Dict[int, Any]]]) – Dictionary mapping categorical features to dictionary mapping codes to categories
- **matrix_vecs** (Optional[List[str]]) – list of objects for matrix encoding, i.e. feature prefixes
- **matrix_feats_per_vec** (Optional[List[str]]) – list of features per vector for matrix encoding, i.e. feature suffixes. Features listed but not present in df will be replaced with NaN.

- **matrix_row_wise** (Optional[bool]) – whether objects encoded as a matrix should be encoded row-wise (i.e. all the features associated with an object are in their own row), or column-wise (i.e. all the features associated with an object are in their own column)
- **tensor_data** (Optional[ndarray]) – data of higher order than a matrix can be passed directly as a numpy array, rather than beign extracted and reshaped from the DataFrame. The array will be saved under matrix data, and this is incompatible with also setting *matrix_vecs*, *matrix_feats_per_vec*, and *matrix_row_wise*. The first dimension of the array must be compatible with the length of the data frame.
- **tensor_name** (Optional[str]) – if *tensor_data* is set, then this is the name that will to the foldfile's metadata.
- **tensor_is_sparse** (bool) – Set to True if the matrix is in sparse COO format and should be densified later on The format expected is *coo_x = sparse.as_coo(x); m = np.vstack((coo_x.data, coo_x.coords))*, where *m* is the tensor passed to *tensor_data*.
- **compression** (Optional[str]) – optional compression argument for h5py, e.g. 'lzf'

Return type None

lumin.data_processing.file_proc.**add_meta_data**(*out_file*, *feats*, *cont_feats*, *cat_feats*, *cat_maps*, *targ_feats*, *wgt_feat=None*, *matrix_vecs=None*, *matrix_feats_per_vec=None*, *matrix_row_wise=None*, *tensor_name=None*, *tensor_shp=None*, *tensor_is_sparse=False*)

Adds meta data to foldfile containing information about the data: feature names, matrix information, etc. *FoldYielder* objects will access this and automatically extract it to save the user from having to manually pass lists of features.

Parameters

- **out_file** (File) – h5py file to save data in
- **feats** (List[str]) – list of all features in data
- **cont_feats** (List[str]) – list of continuous features
- **cat_feats** (List[str]) – list of categorical features
- **cat_maps** (Optional[Dict[str, Dict[int, Any]]]) – Dictionary mapping categorical features to dictionary mapping codes to categories
- **targ_feats** (Union[str, List[str]]) – (list of) target feature(s)
- **wgt_feat** (Optional[str]) – name of weight feature
- **matrix_vecs** (Optional[List[str]]) – list of objects for matrix encoding, i.e. feature prefixes
- **matrix_feats_per_vec** (Optional[List[str]]) – list of features per vector for matrix encoding, i.e. feature suffixes. Features listed but not present in df will be replaced with NaN.
- **matrix_row_wise** (Optional[bool]) – whether objects encoded as a matrix should be encoded row-wise (i.e. all the features associated with an object are in their own row), or column-wise (i.e. all the features associated with an object are in their own column)

- **tensor_name** (Optional[str]) – Name used to refer to the tensor when displaying model information
- **tensor_shp** (Optional[Tuple[int]]) – The shape of the tensor data (exclusing batch dimension)
- **tensor_is_sparse** (bool) – Whether the tensor is sparse (COO format) and should be densified prior to use

Return type None

3.3 lumin.data_processing.hep_proc module

lumin.data_processing.hep_proc.**to_cartesian**(*df*, *vec*, *drop=False*)

Vectoriesed conversion of 3-momenta to Cartesian coordinates inplace, optionally dropping old pT,eta,phi features

Parameters

- **df** (DataFrame) – DataFrame to alter
- **vec** (str) – column prefix of vector components to alter, e.g. 'muon' for columns ['muon_pt', 'muon_phi', 'muon_eta']
- **drop** (bool) – Whether to remove original columns and just keep the new ones

Return type None

lumin.data_processing.hep_proc.**to_pt_eta_phi**(*df*, *vec*, *drop=False*)

Vectorised conversion of 3-momenta to pT,eta,phi coordinates inplace, optionally dropping old px,py,pz features

Parameters

- **df** (DataFrame) – DataFrame to alter
- **vec** (str) – column prefix of vector components to alter, e.g. 'muon' for columns ['muon_px', 'muon_py', 'muon_pz']
- **drop** (bool) – Whether to remove original columns and just keep the new ones

Return type None

lumin.data_processing.hep_proc.**delta_phi**(*arr_a*, *arr_b*)

Vectorised computation of modulo 2pi angular seperation of array of angles b from array of angles a, in range [-pi,pi]

Parameters

- **arr_a** (Union[float, ndarray]) – reference angles
- **arr_b** (Union[float, ndarray]) – final angles

Return type Union[float, ndarray]

Returns angular separation as float or np.ndarray

lumin.data_processing.hep_proc.**twist**(*dphi*, *deta*)

Vectorised computation of twist between vectors (https://arxiv.org/abs/1010.3698)

Parameters

- **dphi** (Union[float, ndarray]) – delta phi separations
- **deta** (Union[float, ndarray]) – delta eta separations

Return type Union[float, ndarray]

Returns angular separation as float or np.ndarray

lumin.data_processing.hep_proc.**add_abs_mom**(*df*, *vec*, *z=True*)

Vectorised computation 3-momenta magnitude, adding new column in place. Currently only works for Cartesian vectors

Parameters

- **df** (DataFrame) – DataFrame to alter
- **vec** (str) – column prefix of vector components, e.g. 'muon' for columns ['muon_px', 'muon_py', 'muon_pz']
- **z** (bool) – whether to consider the z-component of the momenta

Return type None

lumin.data_processing.hep_proc.**add_mass**(*df*, *vec*)

Vectorised computation of mass of 4-vector, adding new column in place.

Parameters

- **df** (DataFrame) – DataFrame to alter
- **vec** (str) – column prefix of vector components, e.g. 'muon' for columns ['muon_px', 'muon_py', 'muon_pz']

Return type None

lumin.data_processing.hep_proc.**add_energy**(*df*, *vec*)

Vectorised computation of energy of 4-vector, adding new column in place.

Parameters

- **df** (DataFrame) – DataFrame to alter
- **vec** (str) – column prefix of vector components, e.g. 'muon' for columns ['muon_px', 'muon_py', 'muon_pz']

Return type None

lumin.data_processing.hep_proc.**add_mt**(*df*, *vec*, *mpt_name='mpt'*)

Vectorised computation of transverse mass of 4-vector with respect to missing transverse momenta, adding new column in place. Currently only works for pT, eta, phi vectors

Parameters

- **df** (DataFrame) – DataFrame to alter
- **vec** (str) – column prefix of vector components, e.g. 'muon' for columns ['muon_px', 'muon_py', 'muon_pz']
- **mpt_name** (str) – column prefix of vector of missing transverse momenta components, e.g. 'mpt' for columns ['mpt_pT', 'mpt_phi']

lumin.data_processing.hep_proc.**get_vecs**(*feats*, *strict=True*)

Filter list of features to get list of 3-momenta defined in the list. Works for both pT, eta, phi and Cartesian coordinates. If strict, return only vectors with all coordinates present in feature list.

Parameters

- **feats** (List[str]) – list of features to filter
- **strict** (bool) – whether to require all 3-momenta components to be present in the list

Return type Set[str]

Returns set of unique 3-momneta prefixes

lumin.data_processing.hep_proc.**fix_event_phi**(*df*, *ref_vec*)

Rotate event in phi such that ref_vec is at phi == 0. Performed inplace. Currently only works on vectors defined in pT, eta, phi

Parameters

- **df** (DataFrame) – DataFrame to alter
- **ref_vec** (str) – column prefix of vector components to use as reference, e.g. 'muon' for columns ['muon_pT', 'muon_eta', 'muon_phi']

Return type None

lumin.data_processing.hep_proc.**fix_event_z**(*df*, *ref_vec*)

Flip event in z-axis such that ref_vec is in positive z-direction. Performed inplace. Works for both pT, eta, phi and Cartesian coordinates.

Parameters

- **df** (DataFrame) – DataFrame to alter
- **ref_vec** (str) – column prefix of vector components to use as reference, e.g. 'muon' for columns ['muon_pT', 'muon_eta', 'muon_phi']

Return type None

lumin.data_processing.hep_proc.**fix_event_y**(*df*, *ref_vec_0*, *ref_vec_1*)

Flip event in y-axis such that ref_vec_1 has a higher py than ref_vec_0. Performed in place. Works for both pT, eta, phi and Cartesian coordinates.

Parameters

- **df** (DataFrame) – DataFrame to alter
- **ref_vec_0** (str) – column prefix of vector components to use as reference 0, e.g. 'muon' for columns ['muon_pT', 'muon_eta', 'muon_phi']
- **ref_vec_1** (str) – column prefix of vector components to use as reference 1, e.g. 'muon' for columns ['muon_pT', 'muon_eta', 'muon_phi']

Return type None

lumin.data_processing.hep_proc.**event_to_cartesian**(*df*, *drop=False*, *ignore=None*)

Convert entire event to Cartesian coordinates, except vectors listed in ignore. Optionally, drop old pT,eta,phi features. Perfomed inplace.

Parameters

- **df** (DataFrame) – DataFrame to alter
- **drop** (bool) – whether to drop old coordinates
- **ignore** (Optional[List[str]]) – vectors to ignore when converting

Return type None

lumin.data_processing.hep_proc.**proc_event**(*df*, *fix_phi=False*, *fix_y=False*, *fix_z=False*, *use_cartesian=False*, *ref_vec_0=None*, *ref_vec_1=None*, *keep_feats=None*, *default_vals=None*)

Process event: Pass data through inplace various conversions and drop uneeded columns. Data expected to consist of vectors defined in pT, eta, phi.

Parameters

- **df** (DataFrame) – DataFrame to alter
- **fix_phi** (bool) – whether to rotate events using fix_event_phi()
- **fix_y** – whether to flip events using fix_event_y()

- **fix_z** – whether to flip events using `fix_event_z()`
- **use_cartesian** – wether to convert vectors to Cartesian coordinates
- **ref_vec_0** (`Optional[str]`) – column prefix of vector components to use as reference (0) for :meth:~lumin.data_prcoessing.hep_proc.fix_event_phi‘, `fix_event_y()`, and `fix_event_z()` e.g. ‘muon’ for columns [‘muon_pT’, ‘muon_eta’, ‘muon_phi’]
- **ref_vec_1** (`Optional[str]`) – column prefix of vector components to use as reference (1) for `fix_event_y()`, e.g. ‘muon’ for columns [‘muon_pT’, ‘muon_eta’, ‘muon_phi’]
- **keep_feats** (`Optional[List[str]]`) – columns to keep which would otherwise be dropped
- **default_vals** (`Optional[List[str]]`) – list of default values which might be used to represent missing vector components. These will be replaced with np.nan.

Return type `None`

lumin.data_processing.hep_proc.**calc_pair_mass**(*df*, *masses*, *feat_map*)

Vectorised computation of invarient mass of pair of particles with given masses, using 3-momenta. Only works for vectors defined in Cartesian coordinates.

Parameters

- **df** (`DataFrame`) – DataFrame vector components
- **masses** (`Union[Tuple[float, float], Tuple[ndarray, ndarray]]`) – tuple of masses of particles (either constant or different pair of masses per pair of particles)
- **feat_map** (`Dict[str, str]`) – dictionary mapping of requested momentum components to the features in df

Return type `ndarray`

Returns np.ndarray of invarient masses

lumin.data_processing.hep_proc.**boost**(*ref_vec*, *boost_vec*, *df=None*, *rescale_boost=False*)

Vectorised boosting of reference vectors along boosting vectors. N.B. Implementation adapted from ROOT (https://root.cern/)

Parameters

- **vec_0** – either (N,4) array of 4-momenta coordinates for starting vector, or prefix name for starting vector, i.e. columns should have names of the form [vec_0]_px, etc.
- **vec_1** – either (N,4) array of 4-momenta coordinates for boosting vector, or prefix name for boosting vector, i.e. columns should have names of the form [vec_1]_px, etc.
- **df** (`Optional[DataFrame]`) – DataFrame with data
- **rescale_boost** (`bool`) – whether to divide the boost vector by its energy

Return type `ndarray`

Returns (N,4) array of boosted vector in Cartesian coordinates

lumin.data_processing.hep_proc.**boost2cm**(*vec*, *df=None*)

Vectorised computation of boosting vector required to boost a vector to its centre-of-mass frame

Parameters

- **vec** (`Union[ndarray, str]`) – either (N,4) array of 4-momenta coordinates for starting vector, or prefix name for starting vector, i.e. columns should have names of the form [vec]_px, etc.

- **df** (Optional[DataFrame]) – DataFrame with data is supplying a string *vec*

Return type ndarray

Returns (N,3) array of boosting vector in Cartesian coordinates

lumin.data_processing.hep_proc.**get_momentum**(*df*, *vec*, *include_E=False*, *as_cart=False*)

Extracts array of 3- or 4-momenta coordinates from DataFrame columns

Parameters

- **df** (DataFrame) – DataFrame with data
- **vec** (str) – prefix name for vector, i.e. columns should have names of the form [vec]_px, etc.
- **as_cart** (bool) – if True will return momenta in Cartesian coordinates

Returns (px, py, pz, (E)) or (pT, phi, eta, (E))

Return type (N, 3|4) array with columns

lumin.data_processing.hep_proc.**cos_delta**(*vec_0*, *vec_1*, *df=None*, *name=None*, *inplace=False*)

Vectorised compututation of the cosine of the angular seperation of *vec_1* from *vec_0* If *vec_** are strings, then columns are extracted from DataFrame *df*. If inplace is True Cosine angle is added a new column to the DataFrame with name *cosdelta_[vec_0]_[vec_1]* or *cosdelta*, unless *name* is set

Parameters

- **vec_0** (Union[ndarray, str]) – either (N,3) array of 3-momenta coordinates for vector 0, or prefix name for vector zero, i.e. columns should have names of the form [vec_0]_px, etc.
- **vec_1** (Union[ndarray, str]) – either (N,3) array of 3-momenta coordinates for vector 1, or prefix name for vector one, i.e. columns should have names of the form [vec_1]_px, etc.
- **df** (Optional[DataFrame]) – DataFrame with data
- **name** (Optional[str]) – if set, will create a new column in df for cosdelta with given name, otherwise will generate a name
- **inplace** (bool) – if True will add new column to df, otherwise will return array of cos_deltas

Return type Union[None, ndarray]

Returns array of cos deltas in not inplace

lumin.data_processing.hep_proc.**delta_r**(*dphi*, *deta*)

Vectorised computation of delta R separation for arrays of delta phi and delta eta (rapidity or pseudorapidity)

Parameters

- **dphi** (Union[float, ndarray]) – delta phi separations
- **deta** (Union[float, ndarray]) – delta eta separations

Return type Union[float, ndarray]

Returns delta R separation as float or np.ndarray

lumin.data_processing.hep_proc.**delta_r_boosted**(*vec_0*, *vec_1*, *ref_vec*, *df=None*, *name=None*, *inplace=False*)

Vectorised compututation of the deltaR seperation of *vec_1* from *vec_0* in the rest-frame of another vector If *vec_** are strings, then columns are extracted from DataFrame *df*. If inplace is True deltaR is added a new column to the DataFrame with name *dR_[vec_0]_[vec_1]_boosted_[ref_vec]* or *dR_boosted*, unless *name* is set

Parameters

- **vec_0** (Union[ndarray, str]) – either (N,4) array of 4-momenta coordinates for vector 0, in Cartesian coordinates or prefix name for vector zero, i.e. columns should have names of the form [vec_0]_px, etc.
- **vec_1** (Union[ndarray, str]) – either (N,4) array of 4-momenta coordinates for vector 1, in Cartesian coordinates or prefix name for vector one, i.e. columns should have names of the form [vec_1]_px, etc.
- **ref_vec** (Union[ndarray, str]) – either (N,4) array of 4-momenta coordinates for the vector in whos rest-frame deltaR should be computed, in Cartesian coordinates or prefix name for reference vector, i.e. columns should have names of the form [ref_vec]_px, etc.
- **df** (Optional[DataFrame]) – DataFrame with data
- **name** (Optional[str]) – if set, will create a new column in df for cosdelta with given name, otherwise will generate a name
- **inplace** (bool) – if True will add new column to df, otherwise will return array of cos_deltas

Return type Union[None, ndarray]

Returns array of boosted deltaR in not inplace

3.4 lumin.data_processing.pre_proc module

lumin.data_processing.pre_proc.**get_pre_proc_pipes**(*norm_in=True*, *norm_out=False*, *pca=False*, *whiten=False*, *with_mean=True*, *with_std=True*, *n_components=None*)

Configure SKLearn Pipelines for processing inputs and targets with the requested transformations.

Parameters

- **norm_in** (bool) – whether to apply StandardScaler to inputs
- **norm_out** (bool) – whether to apply StandardScaler to outputs
- **pca** (bool) – whether to apply PCA to inputs. Perforemed prior to StandardScaler. No dimensionality reduction is applied, purely rotation.
- **whiten** (bool) – whether PCA should whiten inputs.
- **with_mean** (bool) – whether StandardScalers should shift means to 0
- **with_std** (bool) – whether StandardScalers should scale standard deviations to 1
- **n_components** (Optional[int]) – if set, causes PCA to reduce the dimensionality of the input data

Return type Tuple[Pipeline, Pipeline]

Returns Pipeline for input data Pipeline for target data

lumin.data_processing.pre_proc.**fit_input_pipe**(*df*, *cont_feats*, *savename=None*, *input_pipe=None*, *norm_in=True*, *pca=False*, *whiten=False*, *with_mean=True*, *with_std=True*, *n_components=None*)

Fit input pipeline to continuous features and optionally save.

Parameters

- **df** (DataFrame) – DataFrame with data to fit pipeline
- **cont_feats** (Union[str, List[str]]) – (list of) column(s) to use as input data for fitting
- **savename** (Optional[str]) – if set will save the fitted Pipeline to with that name as Pickle (.pkl extension added automatically)
- **input_pipe** (Optional[Pipeline]) – if set will fit, otherwise will instantiate a new Pipeline
- **norm_in** (bool) – whether to apply StandardScaler to inputs. Only used if input_pipe is not set.
- **pca** (bool) – whether to apply PCA to inputs. Perforemed prior to StandardScaler. No dimensionality reduction is applied, purely rotation. Only used if input_pipe is not set.
- **whiten** (bool) – whether PCA should whiten inputs. Only used if input_pipe is not set.
- **with_mean** (bool) – whether StandardScalers should shift means to 0. Only used if input_pipe is not set.
- **with_std** (bool) – whether StandardScalers should scale standard deviations to 1. Only used if input_pipe is not set.
- **n_components** (Optional[int]) – if set, causes PCA to reduce the dimensionality of the input data. Only used if input_pipe is not set.

Return type Pipeline

Returns Fitted Pipeline

lumin.data_processing.pre_proc.**fit_output_pipe**(*df*, *targ_feats*, *savename=None*, *output_pipe=None*, *norm_out=True*)

Fit output pipeline to target features and optionally save. Have you thought about using a y_range for regression instead?

Parameters

- **df** (DataFrame) – DataFrame with data to fit pipeline
- **targ_feats** (Union[str, List[str]]) – (list of) column(s) to use as input data for fitting
- **savename** (Optional[str]) – if set will save the fitted Pipeline to with that name as Pickle (.pkl extension added automatically)
- **output_pipe** (Optional[Pipeline]) – if set will fit, otherwise will instantiate a new Pipeline
- **norm_out** (bool) – whether to apply StandardScaler to outputs . Only used if output_pipe is not set.

Return type Pipeline

Returns Fitted Pipeline

lumin.data_processing.pre_proc.**proc_cats**(*train_df*, *cat_feats*, *val_df=None*, *test_df=None*)

Process categorical features in train_df to be valued 0->cardinality-1. Applied inplace. Applies same transformation to validation and testing data is passed. Will complain if validation or testing sets contain categories which are not present in the training data.

Parameters

- **train_df** (DataFrame) – DataFrame with the training data, which will also be used to specify all the categories to consider
- **cat_feats** (List[str]) – list of columns to use as categorical features
- **val_df** (Optional[DataFrame]) – if set will apply the same category to code mapping to the validation data as was performed on the training data
- **test_df** (Optional[DataFrame]) – if set will apply the same category to code mapping to the testing data as was performed on the training data

Return type Tuple[OrderedDict, OrderedDict]

Returns ordered dictionary mapping categorical features to dictionaries mapping codes to categories ordered dictionary mapping categorical features to their cardinalities

3.5 Module contents

CHAPTER

FOUR

LUMIN.EVALUATION PACKAGE

4.1 Submodules

4.2 lumin.evaluation.ams module

lumin.evaluation.ams.**calc_ams**(*s*, *b*, *br=0*, *unc_b=0*)

Compute Approximate Median Significance (https://arxiv.org/abs/1007.1727)

Parameters

- **s** (float) – signal weight
- **b** (float) – background weight
- **br** (float) – background offset bias
- **unc_b** (float) – fractional systemtatic uncertainty on background

Return type float

Returns Approximate Median Significance if b > 0 else -1

lumin.evaluation.ams.**calc_ams_torch**(*s*, *b*, *br=0*, *unc_b=0*)

Compute Approximate Median Significance (https://arxiv.org/abs/1007.1727) using Tensor inputs

Parameters

- **s** (Tensor) – signal weight
- **b** (Tensor) – background weight
- **br** (float) – background offset bias
- **unc_b** (float) – fractional systemtatic uncertainty on background

Return type Tensor

Returns Approximate Median Significance if b > 0 else 1e-18 * s

lumin.evaluation.ams.**ams_scan_quick**(*df*, *wgt_factor=1*, *br=0*, *syst_unc_b=0*, *pred_name='pred'*, *targ_name='gen_target'*, *wgt_name='gen_weight'*)

Scan across a range of possible prediction thresholds in order to maximise the Approximate Median Significance (https://arxiv.org/abs/1007.1727). Note that whilst this method is quicker than *ams_scan_slow()*, it sufferes from float precison. Not recommended for final evaluation.

Parameters

- **df** (DataFrame) – DataFrame containing prediction data
- **wgt_factor** (float) – factor to reweight signal and background weights

- **br** (float) – background offset bias
- **syst_unc_b** (float) – fractional systemtatic uncertainty on background
- **pred_name** (str) – column to use as predictions
- **targ_name** (str) – column to use as truth labels for signal and background
- **wgt_name** (str) – column to use as weights for signal and background events

Return type Tuple[float, float]

Returns maximum AMS prediction threshold corresponding to maximum AMS

lumin.evaluation.ams.**ams_scan_slow**(*df*, *wgt_factor=1*, *br=0*, *syst_unc_b=0*, *use_stat_unc=False*, *start_cut=0.9*, *min_events=10*, *pred_name='pred'*, *targ_name='gen_target'*, *wgt_name='gen_weight'*, *show_prog=True*)

Scan across a range of possible prediction thresholds in order to maximise the Approximate Median Significance (https://arxiv.org/abs/1007.1727). Note that whilst this method is slower than *ams_scan_quick()*, it does not suffer as much from float precison. Additionally it allows one to account for statistical uncertainty in AMS calculation.

Parameters

- **df** (DataFrame) – DataFrame containing prediction data
- **wgt_factor** (float) – factor to reweight signal and background weights
- **br** (float) – background offset bias
- **syst_unc_b** (float) – fractional systemtatic uncertainty on background
- **use_stat_unc** (bool) – whether to account for the statistical uncertainty on the background
- **start_cut** (float) – minimum prediction to consider; useful for speeding up scan
- **min_events** (int) – minimum number of background unscaled events required to pass threshold
- **pred_name** (str) – column to use as predictions
- **targ_name** (str) – column to use as truth labels for signal and background
- **wgt_name** (str) – column to use as weights for signal and background events
- **show_prog** (bool) – whether to display progress and ETA of scan

Return type Tuple[float, float]

Returns maximum AMS prediction threshold corresponding to maximum AMS

4.3 Module contents

CHAPTER

FIVE

LUMIN.INFERENCE PACKAGE

5.1 Submodules

5.2 lumin.inference.summary_stat module

lumin.inference.summary_stat.**bin_binary_class_pred**(*df*, *max_unc*, *consider_samples=None*, *step_sz=0.001*, *pred_name='pred'*, *sample_name='gen_sample'*, *compact_samples=False*, *class_name='gen_target'*, *add_pure_signal_bin=False*, *max_unc_pure_signal=0.1*, *verbose=True*)

Define bin-edges for binning particle process samples as a function of event class prediction (signal | background) such that the statistical uncertainties on per bin yields are below max_unc for each considered sample.

Parameters

- **df** (DataFrame) – DataFrame containing the data
- **max_unc** (float) – maximum fractional statisitcal uncertainty to allow when defining bins
- **consider_samples** (Optional[List[str]]) – if set, only listed samples are considered when defining bins
- **step_sz** (float) – resolution of scan along event prediction
- **pred_name** (str) – column to use as event class prediction
- **sample_name** (str) – column to use as particle process fo reach event
- **compact_samples** (bool) – if true, will not consider samples when computing bin edges, only the class
- **class_name** (str) – name of column to use as class indicator
- **add_pure_signal_bin** (bool) – if true will attempt to add a bin which oonly contains signal (class 1) if the fractional bin-fill uncertainty would be less than max_unc_pure_signal
- **max_unc_pure_signal** (float) – maximum fractional statisitcal uncertainty to allow when defining pure-signal bins
- **verbose** (bool) – whether to show progress bar

Return type `List[float]`

Returns list of bin edges

5.3 Module contents

CHAPTER

SIX

LUMIN.NN PACKAGE

6.1 Subpackages

6.1.1 lumin.nn.callbacks package

Submodules

lumin.nn.callbacks.adversarial_callbacks module

class lumin.nn.callbacks.adversarial_callbacks.**PivotTraining**(*n_pretrain_main*, *n_pretrain_adv*, *adv_coef*, *adv_model_builder*, *adv_targets*, *adv_update_freq*, *adv_update_on*, *main_pretrain_cb_partials=None*, *adv_pretrain_cb_partials=None*, *adv_train_cb_partials=None*)

Bases: *lumin.nn.callbacks.callback.Callback*

Callback implementation of "Learning to Pivot with Adversarial Networks" (Louppe, Kagan, & Cranmer, 2016) (https://papers.nips.cc/paper/2017/hash/48ab2f9b45957ab574cf005eb8a76760-Abstract.html). The default target data in the *FoldYielder* should be the target data for the main model, and it should contain additional columns for target data for the adversary (names should be passed to the *adv_targets* argument.)

Once training begins, both the main model and the adversary will be pretrained in isolation. Further training of the main model then starts, with the frozen adversary providing a bonus to the loss value if the adversary cannot predict well its targets based on the prediction of the main model. At a set interval (multiples of per batch/fold/epoch), the adversary is refined for 1 epoch with the main model frozen (if per batch, this can take a long time with no progression indicated to the user). States of the model and the adversary are saved to the savepath after both pretraining and further training.

Parameters

- **n_pretrain_main** (int) – number of epochs to pretrain the main model
- **n_pretrain_adv** (int) – number of epochs to pretrain the adversary
- **adv_coef** (float) – relative weighting for the adversarial bonus (lambda in the paper), code assumes a positive value and subtracts adversarial loss from the main loss
- **adv_model_builder** (*ModelBuilder*) – *ModelBuilder* defining the adversary (note that this should not define main_model+adversary)

- **adv_targets** (List[str]) – list of column names in foldfile to use as targets for the adversary
- **adv_update_freq** (int) – sets how often the adversary is refined (e.g. once every *adv_update_freq* ticks)
- **adv_update_on** (str) – str defines the tick for refining the adversary, can be batch, fold, or epoch. The paper refines once for every batch of training data.
- **main_pretrain_cb_partials** (Optional[List[Callable[[], *Callback*]]]) – Optional list of partial callbacks to use when pretraining the main model
- **adv_pretrain_cb_partials** (Optional[List[Callable[[], *Callback*]]]) – Optional list of partial callbacks to use when pretraining the adversary model
- **adv_train_cb_partials** (Optional[List[Callable[[], *Callback*]]]) – Optional list of partial callbacks to use when refining the adversary model

on_batch_begin()
Slices off adversarial and main-model targets. Increments tick if required.

Return type None

on_epoch_begin()
Increments tick if required.

Return type None

on_fold_begin()
Increments tick if required.

Return type None

on_forwards_end()
Applies adversarial bonus to main-model loss

Return type None

on_train_begin()
Pretrains main model and adversary, then prepares for further training. Adds prepends training callbacks with a *TargReplace* instance to grab both the target and pivot data

Return type None

on_train_end()
Save final version of adversary

Return type None

lumin.nn.callbacks.callback module

class lumin.nn.callbacks.callback.**Callback**
Bases: lumin.nn.callbacks.abs_callback.AbsCallback

Base callback class from which other callbacks should inherit.

on_pred_begin()

Return type None

on_train_begin()

Return type None

set_model(*model*)
Sets the callback's model in order to allow the callback to access and adjust model parameters

Parameters **model** (AbsModel) – model to refer to during training

Return type None

set_plot_settings(*plot_settings*)
Sets the plot settings for any plots produced by the callback

Parameters **plot_settings** (*PlotSettings*) – PlotSettings class

Return type None

lumin.nn.callbacks.cyclic_callbacks module

class lumin.nn.callbacks.cyclic_callbacks.**AbsCyclicCallback**(*interp*, *param_range*, *cycle_mult=1*, *decrease_param=False*, *scale=1*, *cycle_save=False*)

Bases: *lumin.nn.callbacks.callback.Callback*

Abstract class for callbacks affecting lr or mom

Parameters

- **interp** (str) – string representation of interpolation function. Either 'linear' or 'cosine'.
- **param_range** (Tuple[float, float]) – minimum and maximum values for parameter
- **cycle_mult** (int) – multiplicative factor for adjusting the cycle length after each cycle. E.g *cycle_mult=1* keeps the same cycle length, *cycle_mult=2* doubles the cycle length after each cycle.
- **decrease_param** (bool) – whether to begin by decreasing the parameter, otherwise begin by increasing it
- **scale** (int) – multiplicative factor for setting the initial number of epochs per cycle. E.g *scale=1* means 1 epoch per cycle, *scale=5* means 5 epochs per cycle.
- **cycle_save** (bool) – if true will save a copy of the model at the end of each cycle. Used for building ensembles from single trainings (e.g. snapshot ensembles)
- **nb** – number of minibatches (iterations) to expect per epoch

on_batch_begin()
Computes the new value for the optimiser parameter and passes it to *_set_param* method

Return type None

on_batch_end()
Increments the callback's progress through the cycle

Return type None

on_epoch_begin()
Ensures the *cycle_end* flag is false when the epoch starts

Return type None

on_train_begin()

Return type None

plot()
Plots the history of the parameter evolution as a function of iterations

Return type None

class lumin.nn.callbacks.cyclic_callbacks.**CycleLR**(*lr_range*, *interp='cosine'*, *cycle_mult=1*, *decrease_param='auto'*, *scale=1*, *cycle_save=False*)
Bases: *lumin.nn.callbacks.cyclic_callbacks.AbsCyclicCallback*

Callback to cycle learning rate during training according to either: cosine interpolation for SGDR https://arxiv.org/abs/1608.03983 or linear interpolation for Smith cycling https://arxiv.org/abs/1506.01186

Parameters

- **lr_range** (Tuple[float, float]) – tuple of initial and final LRs
- **interp** (str) – 'cosine' or 'linear' interpolation
- **cycle_mult** (int) – Multiplicative constant for altering the cycle length after each complete cycle
- **decrease_param** (Union[str, bool]) – whether to increase or decrease the LR (effectively reverses lr_range order), 'auto' selects according to interp
- **scale** (int) – Multiplicative constant for altering the length of a cycle. 1 corresponds to one cycle = one epoch
- **cycle_save** (bool) – if true will save a copy of the model at the end of each cycle. Used for building ensembles from single trainings (e.g. snapshot ensembles)
- **nb** – Number of batches in a epoch

Examples::

```
>>> cosine_lr = CycleLR(lr_range=(0, 2e-3), cycle_mult=2, scale=1,
...                     interp='cosine', nb=100)
>>>
>>> cyclical_lr = CycleLR(lr_range=(2e-4, 2e-3), cycle_mult=1, scale=5,
                          interp='linear', nb=100)
```

class lumin.nn.callbacks.cyclic_callbacks.**CycleMom**(*mom_range*, *interp='cosine'*, *cycle_mult=1*, *decrease_param='auto'*, *scale=1*, *cycle_save=False*)
Bases: *lumin.nn.callbacks.cyclic_callbacks.AbsCyclicCallback*

Callback to cycle momentum (beta 1) during training according to either: cosine interpolation for SGDR https://arxiv.org/abs/1608.03983 or linear interpolation for Smith cycling https://arxiv.org/abs/1506.01186 By default is set to evolve in opposite direction to learning rate, a la https://arxiv.org/abs/1803.09820

Parameters

- **mom_range** (Tuple[float, float]) – tuple of initial and final momenta
- **interp** (str) – 'cosine' or 'linear' interpolation
- **cycle_mult** (int) – Multiplicative constant for altering the cycle length after each complete cycle
- **decrease_param** (Union[str, bool]) – whether to increase or decrease the momentum (effectively reverses mom_range order), 'auto' selects according to interp

- **scale** (int) – Multiplicative constant for altering the length of a cycle. 1 corresponds to one cycle = one epoch
- **cycle_save** (bool) – if true will save a copy of the model at the end of each cycle. Used for building ensembles from single trainings (e.g. snapshot ensembles)
- **nb** – Number of batches in a epoch

Examples::

```
>>> cyclical_mom = CycleMom(mom_range=(0.85 0.95), cycle_mult=1,
...                         scale=5, interp='linear', nb=100)
```

class lumin.nn.callbacks.cyclic_callbacks.**OneCycle**(*lengths*, *lr_range*, *mom_range=(0.85*, *0.95)*, *interp='cosine'*, *cycle_ends_training=True*)

Bases: *lumin.nn.callbacks.cyclic_callbacks.AbsCyclicCallback*

Callback implementing Smith 1-cycle evolution for lr and momentum (beta_1) https://arxiv.org/abs/1803.09820 Default interpolation uses fastai-style cosine function. Automatically triggers early stopping on cycle completion.

Parameters

- **lengths** (Tuple[int, int]) – tuple of number of epochs in first and second stages of cycle
- **lr_range** (Union[Tuple[float, float], Tuple[float, float, float]]) – list of initial and max LRs and optionally a final LR. If only two LRs supplied, then final LR will be zero.
- **mom_range** (Tuple[float, float]) – tuple of initial and final momenta
- **interp** (str) – 'cosine' or 'linear' interpolation
- **cycle_ends_training** (bool) – whether to stop training once the cycle finishes, or continue running at the last LR and momentum

Examples::

```
>>> onecycle = OneCycle(lengths=(15, 30), lr_range=[1e-4, 1e-2],
...                     mom_range=(0.85, 0.95), interp='cosine', nb=100)
```

on_batch_begin()

Computes the new lr and momentum and assigns them to the optimiser

Return type None

plot()

Plots the history of the lr and momentum evolution as a function of iterations

class lumin.nn.callbacks.cyclic_callbacks.**CycleStep**(*frac_reduction*, *patience*, *lengths*, *lr_range*, *mom_range=(0.85*, *0.95)*, *interp='cosine'*, *plot_params=False*)

Bases: *lumin.nn.callbacks.cyclic_callbacks.OneCycle*

Combination of 1-cycle and step decay. Initial 1-cycle finishes, and step decay begins starting from best performing model and optimiser.

Parameters

- **frac_reduction** (float) – fractional reduction of the learning rate with each step

- **patience** (int) – number of epochs to wait before step
- **lengths** (Tuple[int, int]) – OneCycle lengths
- **lr_range** (List[float]) – OneCycle learning rates. Don't have the final LR be too small.
- **mom_range** (Tuple[float, float]) – OneCycle momenta,
- **interp** (str) – Iterpolation mode for OneCycle
- **plot_params** (bool) – If true, will plot the parameter history at the end of training.

on_batch_begin()
Computes the new lr and momentum and assigns them to the optimiser

Return type None

on_epoch_begin()
Increment parameters if stepping

Return type None

on_train_begin()
Reset parameters, and check other callbacks in training.

on_train_end()
Optionally plot the parameter history.

Return type None

plot()
Plots the history of the lr and momentum evolution as a function of iterations

lumin.nn.callbacks.data_callbacks module

class lumin.nn.callbacks.data_callbacks.**BinaryLabelSmooth**(*coefs=0*)
Bases: *lumin.nn.callbacks.callback.Callback*

Callback for applying label smoothing to binary classes, based on https://arxiv.org/abs/1512.00567 Applies smoothing during both training.

Parameters **coefs** (Union[float, Tuple[float, float]]) – Smoothing coefficients: 0->coef[0] 1->1-coef[1]. if passed float, coef[0]=coef[1]

Examples::

```
>>> lbl_smooth = BinaryLabelSmooth(0.1)
>>>
>>> lbl_smooth = BinaryLabelSmooth((0.1, 0.02))
```

on_fold_begin()
Apply smoothing

Return type None

class lumin.nn.callbacks.data_callbacks.**BootstrapResample**(*n_folds*, *bag_each_time=False*, *reweight=True*)
Bases: *lumin.nn.callbacks.callback.Callback*

Callback for bootstrap sampling new training datasets from original training data during (ensemble) training.

Parameters

- **n_folds** (int) – the number of folds present in training *FoldYielder*

- **bag_each_time** (bool) – whether to sample a new set for each sub-epoch or to use the same sample each time
- **reweight** (bool) – whether to reweight the sampleed data to mathch the weight sum (per class) of the original data

Examples::

```
>>> bs_resample BootstrapResample(n_folds=len(train_fy))
```

on_fold_begin()
Resamples training data for new epoch

Return type None

on_train_begin()
Resets internal parameters to prepare for a new training

Return type None

class lumin.nn.callbacks.data_callbacks.**ParametrisedPrediction**(*feats*, *param_feat*, *param_val*)
Bases: *lumin.nn.callbacks.callback.Callback*

Callback for running predictions for a parametersied network (https://arxiv.org/abs/1601.07913); one which has been trained using one of more inputs which represent e.g. different hypotheses for the classes such as an unknown mass of some new particle. In such a scenario, multiple signal datasets could be used for training, with background receiving a random mass. During prediction one then needs to set these parametrisation features all to the same values to evaluat the model's response for that hypothesis. This callback can be passed to the predict method of the model/ensemble to adjust the parametrisation features to the desired values.

Parameters

- **feats** (List[str]) – list of feature names used during training (in the same order)
- **param_feat** (Union[List[str], str]) – the feature name which is to be adjusted, or a list of features to adjust
- **param_val** (Union[List[float], float]) – the value to which to set the paramertisation feature, of the list of values to set the parameterisation features to

Examples::

```
>>> mass_param = ParametrisedPrediction(train_feats, 'res_mass', 300)
>>> model.predict(fold_yeilder, pred_name=f'pred_mass_300', callbacks=[mass_
↪param])
>>>
>>> mass_param = ParametrisedPrediction(train_feats, 'res_mass', 300)
>>> spin_param = ParametrisedPrediction(train_feats, 'spin', 1)
>>> model.predict(fold_yeilder, pred_name=f'pred_mass_300', callbacks=[mass_
↪param, spin_param])
```

on_pred_begin()
Adjusts the data to be passed to the model by setting in place the parameterisation feature to the preset value

Return type None

class lumin.nn.callbacks.data_callbacks.**TargReplace**(*targ_feats*)
Bases: *lumin.nn.callbacks.callback.Callback*

Callback to replace target data with requested data from foldfile, allowing one to e.g. train two models simultaneously with the same inputs but different targets for e.g. adversarial training. At the end of validation epochs, the target data is swapped back to the original target data, to allow for the correct computation of any metrics

Parameters **targ_feats** (List[str]) – list of column names in foldfile to get and horizontally stack to replace target data in current *BatchYielder*

Examples::

```
>>> targ_replace = TargReplace(['is_fake'])
>>> targ_replace = TargReplace(['class', 'is_fake'])
```

on_epoch_end()

Swap target data back at the end of validation epochs

Return type None

on_fold_begin()

Stack new target datasets and replace in target data in current *BatchYielder*

Return type None

lumin.nn.callbacks.loss_callbacks module

class lumin.nn.callbacks.loss_callbacks.**GradClip**(*clip*, *clip_norm=True*)

Bases: *lumin.nn.callbacks.callback.Callback*

Callback for clipping gradients by norm or value.

Parameters

- **clip** (float) – value to clip at
- **clip_norm** (bool) – whether to clip according to norm (*torch.nn.utils.clip_grad_norm_*) or value (*torch.nn.utils.clip_grad_value_*)

Examples::

```
>>> grad_clip = GradClip(1e-5)
```

on_backwards_end()

Clips gradients prior to parameter updates

Return type None

lumin.nn.callbacks.lsuv_init module

This file contains code modfied from https://github.com/ducha-aiki/LSUV-pytorch which is made available under the following BSD 2-Clause "Simplified" Licence: Copyright (C) 2017, Dmytro Mishkin All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met: 1. Redistributions of source code must retain the above copyright

notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT

SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

The Apache Licence 2.0 underwhich the majority of the rest of LUMIN is distributed does not apply to the code within this file.

class lumin.nn.callbacks.lsuv_init.**LsuvInit**(*needed_std=1.0*, *std_tol=0.1*, *max_attempts=10*, *do_orthonorm=True*, *verbose=False*)

Bases: *lumin.nn.callbacks.callback.Callback*

Applies Layer-Sequential Unit-Variance (LSUV) initialisation to model, as per Mishkin & Matas 2016 https://arxiv.org/abs/1511.06422. When training begins for the first time, *Conv1D*, *Conv2D*, *Conv3D*, and *Linear* modules in the model will be LSUV initialised using the BatchYielder inputs. This involves initialising the weights with orthonormal matirces and then iteratively scaling them such that the stadndar deviation of the layer outputs is equal to a desired value, within some tolerance.

Parameters

- **needed_std** (float) – desired standard deviation of layer outputs
- **std_tol** (float) – tolerance for matching standard deviation with target
- **max_attempts** (int) – number of times to attempt weight scaling per layer
- **do_orthonorm** (bool) – whether to apply orthonormal initialisation first, or rescale the exisiting values
- **verbose** (bool) – whether to print out details of the rescaling

Example::

```
>>> lsuv = LsuvInit()
>>>
>>> lsuv = LsuvInit(verbose=True)
>>>
>>> lsuv = LsuvInit(needed_std=0.5, std_tol=0.01, max_attempts=100, do_
↪orthonorm=True)
```

on_fold_begin()

If the LSUV process has yet to run, then it will run using all of the input data provided by the *BatchYielder*

Parameters **by** – BatchYielder providing data for the upcoming epoch

Return type None

on_train_begin()

Sets the callback to initialise the model the first time that *on_epoch_begin* is called.

Return type None

lumin.nn.callbacks.model_callbacks module

class lumin.nn.callbacks.model_callbacks.**SWA**(*start_epoch*, *renewal_period=None*, *update_on_cycle_end=None*, *verbose=False*)

Bases: *lumin.nn.callbacks.callback.Callback*

Callback providing Stochastic Weight Averaging based on (https://arxiv.org/abs/1803.05407) This adapted version allows the tracking of a pair of average models in order to avoid having to hardcode a specific start point for averaging:

- Model average #0 will begin to be tracked start_epoch epochs/cycles after training begins.
- *cycle_since_replacement* is set to 1
- Renewal_period epochs/cycles later, a second average #1 will be tracked.
- At the next renewal period, the performance of #0 and #1 will be compared on data contained in val_fold.
  - **If #0 is better than #1:**
    - #1 is replaced by a copy of the current model
    - cycle_since_replacement is increased by 1
    - renewal_period is multiplied by cycle_since_replacement
  - **Else:**
    - #0 is replaced by #1
    - #1 is replaced by a copy of the current model
    - cycle_since_replacement is set to 1
    - renewal_period is set back to its original value

Additonally, will optionally (default True) lock-in to any cyclical callbacks to only update at the end of a cycle.

Parameters

- **start_epoch** (int) – epoch/cycle to begin averaging
- **renewal_period** (Optional[int]) – How often to check performance of averages, and renew tracking of least performant. If None, will not track a second average.
- **update_on_cycle_end** (Optional[bool]) – Whether to lock in to the cyclic callback and only update at the end of a cycle. Default yes, if cyclic callback present.
- **verbose** (bool) – Whether to print out update information for testing and operation confirmation

Examples::

```
>>> swa = SWA(start_epoch=5, renewal_period=5)
```

get_loss()
Evaluates SWA model and returns loss

Return type float

on_epoch_begin()
Resets loss to prepare for new epoch

Return type None

on_epoch_end()
Checks whether averages should be updated (or reset) and increments counters

Return type None

on_train_begin()
Initialises model variables to begin tracking new model averages

Return type None

lumin.nn.callbacks.monitors module

class lumin.nn.callbacks.monitors.**EarlyStopping**(*patience*, *loss_is_meaned=True*)
Bases: *lumin.nn.callbacks.callback.Callback*

Tracks validation loss during training and terminates training if loss doesn't decrease after *patience* number of epochs. Losses are assumed to be averaged and will be re-averaged over the epoch unless *loss_is_meaned* is false.

Parameters

- **patience** (int) – number of epochs to wait without improvement before stopping training
- **loss_is_meaned** (bool) – if the batch loss value has been averaged over the number of elements in the batch, this should be true; average loss will be computed over all elements in batch. If the batch loss is not an average value, then the average will be computed over the number of batches.

on_epoch_end()
Computes best average validation losses and acts according to it

Return type None

on_train_begin()
Resets variables and prepares for new training

Return type None

class lumin.nn.callbacks.monitors.**SaveBest**(*auto_reload=True*, *loss_is_meaned=True*)
Bases: *lumin.nn.callbacks.callback.Callback*

Tracks validation loss during training and automatically saves a copy of the weights to indicated file whenever validation loss decreases. Losses are assumed to be averaged and will be re-averaged over the epoch unless *loss_is_meaned* is false.

Parameters

- **auto_reload** (bool) – if true, will automatically reload the best model at the end of training
- **loss_is_meaned** (bool) – if the batch loss value has been averaged over the number of elements in the batch, this should be true; average loss will be computed over all elements in batch. If the batch loss is not an average value, then the average will be computed over the number of batches.

on_epoch_end()
Computes best average validation losses and if it is better than the current best, saves a copy of the model which produced it

Return type None

on_train_begin()
Resets variables and prepares for new training

Return type None

on_train_end()
Optionally reload best performing model

Return type None

class lumin.nn.callbacks.monitors.**MetricLogger**(*show_plots=False*, *extra_detail=True*, *loss_is_meaned=True*)
Bases: *lumin.nn.callbacks.callback.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()*.

Parameters

- **loss_names** – List of names of losses which will be passed to the logger in the order in which they will be passed. By convention the first name will be used as the training loss when computing the ratio of training to validation losses
- **n_folds** – Number of folds present in the training data. The logger assumes that one of these folds is for validation, and so 1 training epoch = (n_fold-1) folds.
- **extra_detail** (bool) – Whether to include extra detail plots (loss velocity and training validation ratio), slight slower but potentially useful.

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(*save_best*)

Returns losses and metrics of the (loaded) model

#TODO: extend this to load at specified index

Parameters **save_best** (bool) – if the training used *SaveBest* return results at best point else return the latest values

Return type Dict[str, float]

Returns dictionary of validation loss and metrics

on_batch_end()

Record batch loss

Return type None

on_epoch_begin()

Prepare to track new loss

Return type None

on_epoch_end()

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

Return type None

on_fold_begin()

Prepare to track new loss

Return type None

on_fold_end()

Record training loss for fold

Return type None

on_train_begin()

Prepare for new training

Return type None

print_losses()

Print training and validation losses for the last epoch

Return type None

update_plot()
Updates the plot(s).

TODO: make this faster

Return type None

class lumin.nn.callbacks.monitors.**EpochSaver**
Bases: *lumin.nn.callbacks.callback.Callback*

Callback to save the model at the end of every epoch, regarless of improvement

on_epoch_end()
Save the model at the end of each validation epoch to a new file

on_train_begin()
Reset epoch count

Return type None

lumin.nn.callbacks.opt_callbacks module

class lumin.nn.callbacks.opt_callbacks.**LRFinder**(*lr_bounds=[1e-07, 10], nb=None*)
Bases: *lumin.nn.callbacks.callback.Callback*

Callback class for Smith learning-rate range test (https://arxiv.org/abs/1803.09820)
Parameters

- **nb** (Optional[int]) – number of batches in a epoch
- **lr_bounds** (Tuple[float, float]) – tuple of initial and final LR

get_df()
Returns a DataFrame of LRs and losses

Return type DataFrame

on_batch_end()
Records loss and increments LR

Return type None

on_epoch_begin()
Gets number of batches total on first fold

Return type None

on_train_begin()
Prepares variables and optimiser for new training

Return type None

plot()
Plot the loss as a function of the LR.

Return type None

plot_lr()
Plot the LR as a function of iterations.

Return type None

lumin.nn.callbacks.pred_handlers module

class lumin.nn.callbacks.pred_handlers.**PredHandler**
Bases: *lumin.nn.callbacks.callback.Callback*

Default callback for predictions. Collects predictions over batches and returns them as stacked array

get_preds()

Return type ndarray

on_forwards_end()

Return type None

on_pred_begin()

Return type None

on_pred_end()

Return type None

Module contents

6.1.2 lumin.nn.data package

Submodules

lumin.nn.data.batch_yielder module

class lumin.nn.data.batch_yielder.**BatchYielder**(*inputs*, *bs*, *objective*, *targets=None*, *weights=None*, *shuffle=True*, *use_weights=True*, *bulk_move=True*, *input_mask=None*, *drop_last=True*)
Bases: object

Yields minibatches to model during training. Iteration provides one minibatch as tuple of tensors of inputs, targets, and weights.

Parameters

- **inputs** (Union[ndarray, Tuple[ndarray, ndarray]]) – input array for (sub-)epoch
- **targets** (Optional[ndarray]) – target array for (sub-)epoch
- **bs** (int) – batchsize, number of data to include per minibatch
- **objective** (str) – 'classification', 'multiclass classification', or 'regression'. Used for casting target dtype.
- **weights** (Optional[ndarray]) – Optional weight array for (sub-)epoch
- **shuffle** (bool) – whether to shuffle the data at the beginning of an iteration
- **use_weights** (bool) – if passed weights, whether to actually pass them to the model
- **bulk_move** (bool) – whether to move all data to device at once. Default is true (saves time), but if device has low memory you can set to False.
- **input_mask** (Optional[ndarray]) – optionally only use Boolean-masked inputs
- **drop_last** (bool) – whether to drop the last batch if it does not contain *bs* elements

get_inputs(*on_device=False*)

Return type Union[Tensor, Tuple[Tensor, Tensor]]

lumin.nn.data.fold_yielder module

class lumin.nn.data.fold_yielder.**FoldYielder**(*foldfile*, *cont_feats=None*, *cat_feats=None*, *ignore_feats=None*, *input_pipe=None*, *output_pipe=None*, *yield_matrix=True*, *matrix_pipe=None*)

Bases: object

Interface class for accessing data from foldfiles created by *df2foldfile()*

Parameters

- **foldfile** (Union[str, Path, File]) – filename of hdf5 file or opened hdf5 file
- **cont_feats** (Optional[List[str]]) – list of names of continuous features present in input data, not required if foldfile contains meta data already
- **cat_feats** (Optional[List[str]]) – list of names of categorical features present in input data, not required if foldfile contains meta data already
- **ignore_feats** (Optional[List[str]]) – optional list of input features which should be ignored
- **input_pipe** (Union[str, Pipeline, Path, None]) – optional Pipeline, or filename for pickled Pipeline, which was used for processing the inputs
- **output_pipe** (Union[str, Pipeline, Path, None]) – optional Pipeline, or filename for pickled Pipeline, which was used for processing the targets
- **yield_matrix** (bool) – whether to actually yield matrix data if present
- **matrix_pipe** (Union[str, Pipeline, Path, None]) – preprocessing pipe for matrix data

Examples::

```
>>> fy = FoldYielder('train.h5')
>>>
>>> fy = FoldYielder('train.h5', ignore_feats=['phi'], input_pipe='input_
↪pipe.pkl')
>>>
>>> fy = FoldYielder('train.h5', input_pipe=input_pipe, matrix_pipe=matrix_
↪pipe)
>>>
>>> fy = FoldYielder('train.h5', input_pipe=input_pipe, yield_matrix=False)
```

add_ignore(*feats*)

Add features to ignored features.

Parameters **feats** (Union[str, List[str]]) – list of feature names to ignore

Return type None

add_input_pipe(*input_pipe*)

Adds an input pipe to the FoldYielder for use when deprocessing data

Parameters **input_pipe** (Union[str, Pipeline]) – Pipeline which was used for preprocessing the input data or name of pkl file containing Pipeline

Return type None

add_input_pipe_from_file(*name*)
Adds an input pipe from a pkl file to the FoldYielder for use when deprocessing data

Parameters **name** (Union[str, Path]) – name of pkl file containing Pipeline which was used for preprocessing the input data

Return type None

add_matrix_pipe(*matrix_pipe*)
Adds an matrix pipe to the FoldYielder for use when deprocessing data

Warning: Deprocessing matrix data is not yet implemented

Parameters **matrix_pipe** (Union[str, Pipeline]) – Pipeline which was used for preprocessing the input data or name of pkl file containing Pipeline

Return type None

add_matrix_pipe_from_file(*name*)
Adds an matrix pipe from a pkl file to the FoldYielder for use when deprocessing data

Parameters **name** (str) – name of pkl file containing Pipeline which was used for preprocessing the matrix data

Return type None

add_output_pipe(*output_pipe*)
Adds an output pipe to the FoldYielder for use when deprocessing data

Parameters **output_pipe** (Union[str, Pipeline]) – Pipeline which was used for preprocessing the target data or name of pkl file containing Pipeline

Return type None

add_output_pipe_from_file(*name*)
Adds an output pipe from a pkl file to the FoldYielder for use when deprocessing data

Parameters **name** (Union[str, Path]) – name of pkl file containing Pipeline which was used for preprocessing the target data

Return type None

close()
Closes the foldfile

Return type None

columns()
Returns list of columns present in foldfile

Return type List[str]

Returns list of columns present in foldfile

get_column(*column*, *n_folds=None*, *fold_idx=None*, *add_newaxis=False*)
Load column (h5py group) from foldfile. Used for getting arbitrary data which isn't automatically grabbed by other methods.

Parameters

- **column** (str) – name of h5py group to get

- **n_folds** (Optional[int]) – number of folds to get data from. Default all folds. Not compatable with fold_idx
- **fold_idx** (Optional[int]) – Only load group from a single, specified fold. Not compatable with n_folds
- **add_newaxis** (bool) – whether expand shape of returned data if data shape is ()

Return type Optional[ndarray]

Returns Numpy array of column data

get_data(*n_folds=None*, *fold_idx=None*)

Get data for single, specified fold or several of folds. Data consists of dictionary of inputs, targets, and weights. Does not account for ignored features. Inputs are passed through np.nan_to_num to deal with nans and infs.

Parameters

- **n_folds** (Optional[int]) – number of folds to get data from. Default all folds. Not compatable with fold_idx
- **fold_idx** (Optional[int]) – Only load group from a single, specified fold. Not compatable with n_folds

Return type Dict[str, ndarray]

Returns tuple of inputs, targets, and weights as Numpy arrays

get_data_count(*idxs*)

Returns total number of data entries in requested folds

Parameters **idxs** (Union[int, List[int]]) – list of indices to check

Return type int

Returns Total number of entries in the folds

get_df(*pred_name='pred'*, *targ_name='targets'*, *wgt_name='weights'*, *n_folds=None*, *fold_idx=None*, *inc_inputs=False*, *inc_ignore=False*, *deprocess=False*, *verbose=True*, *suppress_warn=False*, *nan_to_num=False*, *inc_matrix=False*)

Get a Pandas DataFrameof the data in the foldfile. Will add columns for inputs (if requested), targets, weights, and predictions (if present)

Parameters

- **pred_name** (str) – name of prediction group
- **targ_name** (str) – name of target group
- **wgt_name** (str) – name of weight group
- **n_folds** (Optional[int]) – number of folds to get data from. Default all folds. Not compatable with fold_idx
- **fold_idx** (Optional[int]) – Only load group from a single, specified fold. Not compatable with n_folds
- **inc_inputs** (bool) – whether to include input data
- **inc_ignore** (bool) – whether to include ignored features
- **deprocess** (bool) – whether to deprocess inputs and targets if pipelines have been
- **verbose** (bool) – whether to print the number of datapoints loaded

- **suppress_warn** (bool) – whether to supress the warning about missing columns
- **nan_to_num** (bool) – whether to pass input data through *np.nan_to_num*
- **inc_matrix** (bool) – whether to include flattened matrix data in output, if present

Return type DataFrame

Returns Pandas DataFrame with requested data

get_fold(*idx*)

Get data for single fold. Data consists of dictionary of inputs, targets, and weights. Accounts for ignored features. Inputs, except for matrix data, are passed through np.nan_to_num to deal with nans and infs.

Parameters **idx** (int) – fold index to load

Return type Dict[str, ndarray]

Returns tuple of inputs, targets, and weights as Numpy arrays

get_ignore()

Returns list of ignored features

Return type List[str]

Returns Features removed from training data

get_use_cat_feats()

Returns list of categorical features which will be present in training data, accounting for ignored features.

Return type List[str]

Returns List of categorical features

get_use_cont_feats()

Returns list of continuous features which will be present in training data, accounting for ignored features.

Return type List[str]

Returns List of continuous features

save_fold_pred(*pred*, *fold_idx*, *pred_name='pred'*)

Save predictions for given fold as a new column in the foldfile

Parameters

- **pred** (ndarray) – array of predictions in the same order as data appears in the file
- **fold_idx** (int) – index for fold
- **pred_name** (str) – name of column to save predictions under

Return type None

class lumin.nn.data.fold_yielder.**HEPAugFoldYielder**(*foldfile*, *cont_feats=None*, *cat_feats=None*, *ignore_feats=None*, *targ_feats=None*, *rot_mult=2*, *random_rot=False*, *reflect_x=False*, *reflect_y=True*, *reflect_z=True*, *train_time_aug=True*, *test_time_aug=True*, *input_pipe=None*, *output_pipe=None*, *yield_matrix=True*, *matrix_pipe=None*)

Bases: *lumin.nn.data.fold_yielder.FoldYielder*

Specialised version of *FoldYielder* providing HEP specific data augmetation at train and test time.

Parameters

- **foldfile** (Union[str, Path, File]) – filename of hdf5 file or opened hdf5 file
- **cont_feats** (Optional[List[str]]) – list of names of continuous features present in input data, not required if foldfile contains meta data already
- **cat_feats** (Optional[List[str]]) – list of names of categorical features present in input data, not required if foldfile contains meta data already
- **ignore_feats** (Optional[List[str]]) – optional list of input features which should be ignored
- **targ_feats** (Optional[List[str]]) – optional list of target features to also be transformed
- **rot_mult** (int) – number of rotations of event in phi to make at test-time (currently must be even). Greater than zero will also apply random rotations during train-time
- **random_rot** (bool) – whether test-time rotation angles should be random or in steps of 2pi/rot_mult
- **reflect_x** (bool) – whether to reflect events in x axis at train and test time
- **reflect_y** (bool) – whether to reflect events in y axis at train and test time
- **reflect_z** (bool) – whether to reflect events in z axis at train and test time
- **train_time_aug** (bool) – whether to apply augmentations at train time
- **test_time_aug** (bool) – whether to apply augmentations at test time
- **input_pipe** (Optional[Pipeline]) – optional Pipeline, or filename for pickled Pipeline, which was used for processing the inputs
- **output_pipe** (Optional[Pipeline]) – optional Pipeline, or filename for pickled Pipeline, which was used for processing the targets
- **yield_matrix** (bool) – whether to actually yield matrix data if present
- **matrix_pipe** (Union[str, Pipeline, None]) – preprocessing pipe for matrix data

Examples::

```
>>> fy = HEPAugFoldYielder('train.h5',
...                        cont_feats=['pT','eta','phi','mass'],
...                        rot_mult=2, reflect_y=True, reflect_z=True,
...                        input_pipe='input_pipe.pkl')
```

get_fold(*idx*)

Get data for single fold applying random train-time data augmentaion. Data consists of dictionary of inputs, targets, and weights. Accounts for ignored features. Inputs, except for matrix data, are passed through np.nan_to_num to deal with nans and infs.

Parameters **idx** (int) – fold index to load

Return type Dict[str, ndarray]

Returns tuple of inputs, targets, and weights as Numpy arrays

get_test_fold(*idx*, *aug_idx*)

Get test data for single fold applying test-time data augmentaion. Data consists of dictionary of inputs, targets, and weights. Accounts for ignored features. Inputs, except for matrix data, are passed through np.nan_to_num to deal with nans and infs.

Parameters

- **idx** (int) – fold index to load
- **aug_idx** (int) – index for the test-time augmentaion (ignored if random test-time augmentation requested)

Return type Dict[str, ndarray]

Returns tuple of inputs, targets, and weights as Numpy arrays

Module contents

6.1.3 lumin.nn.ensemble package

Submodules

lumin.nn.ensemble.ensemble module

class lumin.nn.ensemble.ensemble.**Ensemble**(*input_pipe=None*, *output_pipe=None*, *model_builder=None*)

Bases: lumin.nn.ensemble.abs_ensemble.AbsEnsemble

Standard class for building an ensemble of collection of trained networks producedd by fold_train_ensemble() Input and output pipelines can be added. to provide easy saving and loaded of exported ensembles. Currently, the input pipeline is not used, so input data is expected to be preprocessed. However the output pipeline will be used to deprocess model predictions.

Once instanciated, lumin.nn.ensemble.ensemble.Ensemble.build_ensemble() or :meth:load should be called. Alternatively, class_methods *lumin.nn.ensemble.ensemble.Ensemble.from_save()* or *lumin.nn.ensemble.ensemble.Ensemble.from_results()* may be used.

TODO: check whether model_builder is necessary here # TODO: Standardise pipeline treatment: currently inputs not processed, but outputs are

Parameters

- **input_pipe** (Optional[Pipeline]) – Optional input pipeline, alternatively call *lumin.nn.ensemble.ensemble.Ensemble.add_input_pipe()*

- **output_pipe** (Optional[Pipeline]) – Optional output pipeline, alternatively call *lumin.nn.ensemble.ensemble.Ensemble.add_output_pipe()*
- **model_builder** (Optional[*ModelBuilder*]) – Optional *ModelBuilder* for constructing models from saved weights.

Examples::

```
>>> ensemble = Ensemble()
>>>
>>> ensemble = Ensemble(input_pipe, output_pipe, model_builder)
```

add_input_pipe(*pipe*)
Add input pipeline for saving

Parameters **pipe** (Pipeline) – pipeline used for preprocessing input data

Return type None

add_output_pipe(*pipe*)
Add output pipeline for saving

Parameters **pipe** (Pipeline) – pipeline used for preprocessing target data

Return type None

export2onnx(*base_name*, *bs=1*)
Export all *Model* contained in *Ensemble* to ONNX format. Note that ONNX expects a fixed batch size (bs) which is the number of datapoints your wish to pass through the model concurrently.

Parameters

- **base_name** (str) – Exported models will be called {base_name}_{model_num}.onnx
- **bs** (int) – batch size for exported models

Return type None

export2tfpb(*base_name*, *bs=1*)
Export all *Model* contained in *Ensemble* to Tensorflow ProtocolBuffer format, via ONNX. Note that ONNX expects a fixed batch size (bs) which is the number of datapoints your wish to pass through the model concurrently.

Parameters

- **base_name** (str) – Exported models will be called {base_name}_{model_num}.pb
- **bs** (int) – batch size for exported models

Return type None

classmethod from_models(*models*, *weights=None*, *results=None*, *input_pipe=None*, *output_pipe=None*, *model_builder=None*)
Instantiate *Ensemble* from a list of Model, and the associated *ModelBuilder*.

Parameters

- **models** (List[AbsModel]) – list of Model
- **weights** (Union[ndarray, List[float], None]) – Optional list of weights, otherwise models will be weighted uniformly
- **results** (Optional[List[Dict[str, float]]]) – Optional results saved/returned by fold_train_ensemble()

- **input_pipe** (Optional[Pipeline]) – Optional input pipeline, alternatively call *lumin.nn.ensemble.ensemble.Ensemble.add_input_pipe()*
- **output_pipe** (Optional[Pipeline]) – Optional output pipeline, alternatively call *lumin.nn.ensemble.ensemble.Ensemble.add_output_pipe()*
- **model_builder** (Optional[*ModelBuilder*]) – Optional *ModelBuilder* for constructing models from saved weights.

Return type AbsEnsemble

Returns Built *Ensemble*

Examples::

```
>>> ensemble = Ensemble.from_models(models)
>>>
>>> ensemble = Ensemble.from_models(models, weights)
>>>
>>> ensemble = Ensemble(models, weights, input_pipe, output_pipe, model_
↪builder)
```

classmethod from_results(*results*, *size*, *model_builder*, *metric='loss'*, *weighting='reciprocal'*, *higher_metric_better=False*, *snapshot_args=None*, *verbose=True*)

Instantiate *Ensemble* from a outputs of fold_train_ensemble(). If cycle models are loaded, then only uniform weighting between models is supported.

Parameters

- **results** (List[Dict[str, float]]) – results saved/returned by fold_train_ensemble()
- **size** (int) – number of models to load as ranked by metric
- **model_builder** (*ModelBuilder*) – *ModelBuilder* used for building *Model* from saved models
- **metric** (str) – metric name listed in results to use for ranking and weighting trained models
- **weighting** (str) – 'reciprocal' or 'uniform' how to weight model predictions during predicition. 'reciprocal' = models weighted by 1/metric 'uniform' = models treated with equal weighting
- **higher_metric_better** (bool) – whether metric should be maximised or minimised
- **snapshot_args** (Optional[Dict[str, Any]]) – Dictionary potentially containing: 'cycle_losses': returned/save by fold_train_ensemble() when using an *AbsCyclicCallback* 'patience': patience value that was passed to fold_train_ensemble() 'n_cycles': number of cycles to load per model 'load_cycles_only': whether to only load cycles, or also the best performing model 'weighting_pwr': weight cycles according to (n+1)**weighting_pwr, where n is the number of cycles loaded so far.

  Models are loaded youngest to oldest
- **verbose** (bool) – whether to print out information of models loaded

Return type AbsEnsemble

Returns Built *Ensemble*

Examples::

```
>>> ensemble = Ensemble.from_results(results, 10, model_builder,
...                                  location=Path('train_weights'))
>>>
>>> ensemble = Ensemble.from_results(
...     results, 1,  model_builder,
...     location=Path('train_weights'),
...     snapshot_args={'cycle_losses':cycle_losses,
...                    'patience':patience,
...                    'n_cycles':8,
...                    'load_cycles_only':True,
...                    'weighting_pwr':0})
```

classmethod from_save(*name*)

Instantiate *Ensemble* from a saved *Ensemble*

Parameters **name** (str) – base filename of ensemble

Return type AbsEnsemble

Returns Loaded *Ensemble*

Examples::

```
>>> ensemble = Ensemble.from_save('weights/ensemble')
```

get_feat_importance(*fy*, *bs=None*, *eval_metric=None*, *savename=None*, *plot_settings=<lumin.plotting.plot_settings.PlotSettings object>*)

Call *get_ensemble_feat_importance()*, passing this *Ensemble* and provided arguments

Parameters

- **fy** (*FoldYielder*) – *FoldYielder* interfacing to data on which to evaluate importance
- **bs** (Optional[int]) – If set, will evaluate model in batches of data, rather than all at once
- **eval_metric** (Optional[*EvalMetric*]) – Optional EvalMetric to use to quantify performance in place of loss
- **savename** (Optional[str]) – Optional name of file to which to save the plot of feature importances
- **plot_settings** (*PlotSettings*) – *PlotSettings* class to control figure appearance

Return type DataFrame

load(*name*)

Load an instantiated *Ensemble* with weights and *Model* from save.

Arguments; name: base name for saved objects

Examples::

```
>>> ensemble.load('weights/ensemble')
```

Return type None

static load_trained_model(*model_idx*, *model_builder*, *name='train_weights/train_'*)
Load trained model from save file of the form *{name}{model_idx}.h5*

Arguments model_idx: index of model to load model_builder: *ModelBuilder* used to build the model name: base name of file from which to load model

Return type *Model*

Returns Model loaded from save

predict(*inputs*, *n_models=None*, *pred_name='pred'*, *pred_cb=<lumin.nn.callbacks.pred_handlers.PredHandler object>*, *cbs=None*, *verbose=True*, *bs=None*, *auto_deprocess=False*)
Apply ensemble to inputed data and compute predictions.

Parameters

- **inputs** (Union[ndarray, *FoldYielder*, List[ndarray]]) – input data as Numpy array, Pandas DataFrame, or tensor on device, or *FoldYielder* interfacing to data
- **as_np** – whether to return predictions as Numpy array (otherwise tensor) if inputs are a Numpy array, Pandas DataFrame, or tensor
- **pred_name** (str) – name of group to which to save predictions if inputs are a *FoldYielder*
- **pred_cb** (*PredHandler*) – *PredHandler* callback to determin how predictions are computed. Default simply returns the model predictions. Other uses could be e.g. running argmax on a multiclass classifier
- **cbs** (Optional[List[AbsCallback]]) – list of any instantiated callbacks to use during prediction
- **bs** (Optional[int]) – if not *None*, will run prediction in batches of specified size to save of memory
- **auto_deprocess** (bool) – if true and ensemble has an output_pipe, will inverse-transform predictions

Return type Union[None, ndarray]

Returns if inputs are a Numpy array, Pandas DataFrame, or tensor, will return predicitions as either array or tensor

save(*name*, *feats=None*, *overwrite=False*)
Save ensemble and associated objects

Parameters

- **name** (str) – base name for saved objects
- **feats** (Optional[Any]) – optional list of input features
- **overwrite** (bool) – if existing objects are found, whether to overwrite them

Examples::

```
>>> ensemble.save('weights/ensemble')
>>>
>>> ensemble.save('weights/ensemble', ['pt','eta','phi'])
```

Return type None

Module contents

6.1.4 lumin.nn.interpretation package

Submodules

lumin.nn.interpretation.features module

lumin.nn.interpretation.features.**get_nn_feat_importance**(*model*, *fy*, *bs=None*, *eval_metric=None*, *pb_parent=None*, *plot=True*, *savename=None*, *settings=<lumin.plotting.plot_settings.PlotSettings object>*)

Compute permutation importance of features used by a *Model* on provided data using either loss or an EvalMetric to quantify performance. Returns bootstrapped mean importance from sample constructed by computing importance for each fold in fy.

Parameters

- **model** (AbsModel) – *Model* to use to evaluate feature importance
- **fy** (*FoldYielder*) – *FoldYielder* interfacing to data used to train model
- **bs** (Optional[int]) – If set, will evaluate model in batches of data, rather than all at once
- **eval_metric** (Optional[*EvalMetric*]) – Optional EvalMetric to use to quantify performance in place of loss
- **pb_parent** (Optional[ConsoleMasterBar]) – Not used if calling method directly
- **plot** (bool) – whether to plot resulting feature importances
- **savename** (Optional[str]) – Optional name of file to which to save the plot of feature importances
- **settings** (*PlotSettings*) – *PlotSettings* class to control figure appearance

Return type DataFrame

Returns Pandas DataFrame containing mean importance and associated uncertainty for each feature

Examples::

```
>>> fi = get_nn_feat_importance(model, train_fy)
>>>
>>> fi = get_nn_feat_importance(model, train_fy, savename='feat_import')
>>>
>>> fi = get_nn_feat_importance(model, train_fy,
...                             eval_metric=AMS(n_total=100000))
```

lumin.nn.interpretation.features.**get_ensemble_feat_importance**(*ensemble*, *fy*, *bs=None*, *eval_metric=None*, *savename=None*, *settings=<lumin.plotting.plot_settings.PlotSe object>*)

Compute permutation importance of features used by an *Ensemble* on provided data using either loss or an EvalMetric to quantify performance. Returns bootstrapped mean importance from sample constructed by computing importance for each *Model* in ensemble.

Parameters

- **ensemble** (AbsEnsemble) – *Ensemble* to use to evaluate feature importance
- **fy** (*FoldYielder*) – *FoldYielder* interfacing to data used to train models in ensemble
- **bs** (Optional[int]) – If set, will evaluate model in batches of data, rather than all at once
- **eval_metric** (Optional[*EvalMetric*]) – Optional EvalMetric to use to quantify performance in place of loss
- **savename** (Optional[str]) – Optional name of file to which to save the plot of feature importances
- **settings** (*PlotSettings*) – *PlotSettings* class to control figure appearance

Return type DataFrame

Returns Pandas DataFrame containing mean importance and associated uncertainty for each feature

Examples::

```
>>> fi = get_ensemble_feat_importance(ensemble, train_fy)
>>>
>>> fi = get_ensemble_feat_importance(ensemble, train_fy
...                                   savename='feat_import')
>>>
>>> fi = get_ensemble_feat_importance(ensemble, train_fy,
...                                   eval_metric=AMS(n_total=100000))
```

TODO: Weight models

Module contents

6.1.5 lumin.nn.losses package

Submodules

lumin.nn.losses.advanced_losses module

class lumin.nn.losses.advanced_losses.**WeightedFractionalMSE**(*weight=None*)

Bases: torch.nn.modules.loss.MSELoss

Class for computing the Mean fractional Squared-Error loss (<Delta^2/true>) with optional weights per prediction. For compatability with using basic PyTorch losses, weights are passed during initialisation rather than when computing the loss.

Parameters **weight** (Optional[Tensor]) – sample weights as PyTorch Tensor, to be used with data to be passed when computing the loss

Examples::

```
>>> loss = WeightedFractionalMSE()
>>>
>>> loss = WeightedFractionalMSE(weights)
```

forward(*input*, *target*)

Evaluate loss for given predictions

Parameters

- **input** (Tensor) – prediction tensor
- **target** (Tensor) – target tensor

Return type Tensor

Returns (weighted) loss

class lumin.nn.losses.advanced_losses.**WeightedBinnedHuber**(*perc*, *bins*, *mom=0.1*, *weight=None*)

Bases: torch.nn.modules.loss.MSELoss

Class for computing the Huberised Mean Squared-Error loss (<Delta^2>) with optional weights per prediction. Losses soft-clamped with Huber like term above adaptive percentile in bins of the target. The thresholds used to transition from MSE to MAE per bin are initialised using the first batch of data as the value of the specified percentile in each bin, subsequently, the thresholds evolve according to: T <- (1-mom)*T + mom*T_batch, where T_batch are the percentiles comuted on the current batch, and mom(emtum) lies between [0,1]

For compatability with using basic PyTorch losses, weights are passed during initialisation rather than when computing the loss.

Parameters

- **perc** (float) – quantile of data in each bin above which to use MAE rather than MSE
- **bins** (Tensor) – tensor of edges for the binning of the target data
- **mom** – momentum for the running average of the thresholds
- **weight** (Optional[Tensor]) – sample weights as PyTorch Tensor, to be used with data to be passed when computing the loss

Examples::

```
>>> loss = WeightedBinnedHuber(perc=0.68)
>>>
>>> loss = WeightedBinnedHuber(perc=0.68, weights=weights)
```

forward(*input*, *target*)

Evaluate loss for given predictions

Parameters

- **input** (Tensor) – prediction tensor
- **target** (Tensor) – target tensor

Return type Tensor

Returns (weighted) loss

class lumin.nn.losses.advanced_losses.**WeightedFractionalBinnedHuber**(*perc*, *bins*, *mom=0.1*, *weight=None*)

Bases: *lumin.nn.losses.advanced_losses.WeightedBinnedHuber*

Class for computing the Huberised Mean fractional Squared-Error loss (<Delta^2/true>) with optional weights per prediction. Losses soft-clamped with Huber like term above adaptive percentile in bins of the target. The thresholds used to transition from MSE to MAE per bin are initialised using the first batch of data as the value of the specified percentile in each bin, subsequently, the thresholds evolve according to: T <- (1-mom)*T + mom*T_batch, where T_batch are the percentiles comuted on the current batch, and mom(emtum) lies between [0,1]

For compatability with using basic PyTorch losses, weights are passed during initialisation rather than when computing the loss.

Parameters

- **perc** (float) – quantile of data in each bin above which to use MAE rather than MSE
- **bins** (Tensor) – tensor of edges for the binning of the target data
- **mom** – momentum for the running average of the thresholds
- **weight** (Optional[Tensor]) – sample weights as PyTorch Tensor, to be used with data to be passed when computing the loss

forward(*input*, *target*)

Evaluate loss for given predictions

Parameters

- **input** (Tensor) – prediction tensor
- **target** (Tensor) – target tensor

Return type Tensor

Returns (weighted) loss

lumin.nn.losses.basic_weighted module

class lumin.nn.losses.basic_weighted.**WeightedMSE**(*weight=None*)

Bases: torch.nn.modules.loss.MSELoss

Class for computing Mean Squared-Error loss with optional weights per prediction. For compatability with using basic PyTorch losses, weights are passed during initialisation rather than when computing the loss.

Parameters **weight** (Optional[Tensor]) – sample weights as PyTorch Tensor, to be used with data to be passed when computing the loss

Examples::

```
>>> loss = WeightedMSE()
>>>
>>> loss = WeightedMSE(weights)
```

forward(*input*, *target*)

Evaluate loss for given predictions

Parameters

- **input** (Tensor) – prediction tensor
- **target** (Tensor) – target tensor

Return type Tensor

Returns (weighted) loss

class lumin.nn.losses.basic_weighted.**WeightedMAE**(*weight=None*)
Bases: torch.nn.modules.loss.L1Loss

Class for computing Mean Absolute-Error loss with optional weights per prediction. For compatability with using basic PyTorch losses, weights are passed during initialisation rather than when computing the loss.

Parameters **weight** (Optional[Tensor]) – sample weights as PyTorch Tensor, to be used with data to be passed when computing the loss

Examples::

```
>>> loss = WeightedMAE()
>>>
>>> loss = WeightedMAE(weights)
```

forward(*input*, *target*)
Evaluate loss for given predictions

Parameters

- **input** (Tensor) – prediction tensor
- **target** (Tensor) – target tensor

Return type Tensor

Returns (weighted) loss

class lumin.nn.losses.basic_weighted.**WeightedCCE**(*weight=None*)
Bases: torch.nn.modules.loss.NLLLoss

Class for computing Categorical Cross-Entropy loss with optional weights per prediction. For compatability with using basic PyTorch losses, weights are passed during initialisation rather than when computing the loss.

Parameters **weight** (Optional[Tensor]) – sample weights as PyTorch Tensor, to be used with data to be passed when computing the loss

Examples::

```
>>> loss = WeightedCCE()
>>>
>>> loss = WeightedCCE(weights)
```

forward(*input*, *target*)
Evaluate loss for given predictions

Parameters

- **input** (Tensor) – prediction tensor
- **target** (Tensor) – target tensor

Return type Tensor

Returns (weighted) loss

lumin.nn.losses.hep_losses module

class lumin.nn.losses.hep_losses.**SignificanceLoss**(*weight*, *sig_wgt=<class 'float'>*, *bkg_wgt=<class 'float'>*, *func=typing.Callable[[torch.Tensor, torch.Tensor], torch.Tensor]*)

Bases: torch.nn.modules.module.Module

General class for implementing significance-based loss functions, e.g. Asimov Loss (https://arxiv.org/abs/1806.00322). For compatability with using basic PyTorch losses, event weights are passed during initialisation rather than when computing the loss.

Parameters

- **weight** (Tensor) – sample weights as PyTorch Tensor, to be used with data to be passed when computing the loss
- **sig_wgt** – total weight of signal events
- **bkg_wgt** – total weight of background events
- **func** – callable which returns a float based on signal and background weights

Examples::

```
>>> loss = SignificanceLoss(weight, sig_weight=sig_weight,
...                         bkg_weight=bkg_weight, func=calc_ams_torch)
>>>
>>> loss = SignificanceLoss(weight, sig_weight=sig_weight,
...                         bkg_weight=bkg_weight,
...                         func=partial(calc_ams_torch, br=10))
```

forward(*input*, *target*)

Evaluate loss for given predictions

Parameters

- **input** (Tensor) – prediction tensor
- **target** (Tensor) – target tensor

Return type Tensor

Returns (weighted) loss

Module contents

6.1.6 lumin.nn.metrics package

Submodules

lumin.nn.metrics.class_eval module

class lumin.nn.metrics.class_eval.**AMS**(*n_total*, *wgt_name*, *br=0*, *syst_unc_b=0*, *use_quick_scan=True*, *name='AMS'*, *main_metric=True*)

Bases: *lumin.nn.metrics.eval_metric.EvalMetric*

Class to compute maximum Approximate Median Significance (https://arxiv.org/abs/1007.1727) using classifier which directly predicts the class of data in a binary classifiaction problem. AMS is computed on a single fold of

data provided by a *FoldYielder* and automatically reweights data by event multiplicity to account missing weights.

Parameters

- **n_total** (int) – total number of events in entire data set
- **wgt_name** (str) – name of weight group in fold file to use. N.B. if you have reweighted to balance classes, be sure to use the un-reweighted weights.
- **br** (float) – constant bias offset for background yield
- **syst_unc_b** (float) – fractional systematic uncertainty on background yield
- **use_quick_scan** (bool) – whether to optimise AMS by the *ams_scan_quick()* method (fast but suffers floating point precision) if False use *ams_scan_slow()* (slower but more accurate)
- **name** (Optional[str]) – optional name for metric, otherwise will be 'AMS'
- **main_metric** (bool) – whether this metic should be treated as the primary metric for SaveBest and EarlyStopping Will automatically set the first EvalMetric to be main if multiple primary metrics are submitted

Examples::

```
>>> ams_metric = AMS(n_total=250000, br=10, wgt_name='gen_orig_weight')
>>>
>>> ams_metric = AMS(n_total=250000, syst_unc_b=0.1,
...                  wgt_name='gen_orig_weight', use_quick_scan=False)
```

evaluate()

Compute maximum AMS on fold using provided predictions.

Return type float

Returns Maximum AMS computed on reweighted data from fold

class lumin.nn.metrics.class_eval.**MultiAMS**(*n_total*, *wgt_name*, *targ_name*, *zero_preds*, *one_preds*, *br=0*, *syst_unc_b=0*, *use_quick_scan=True*, *name='AMS'*, *main_metric=True*)

Bases: *lumin.nn.metrics.eval_metric.EvalMetric*

Class to compute maximum Approximate Median Significance (https://arxiv.org/abs/1007.1727) using classifier which predicts the class of data in a multiclass classifiaction problem which can be reduced to a binary classification problem AMS is computed on a single fold of data provided by a *FoldYielder* and automatically reweights data by event multiplicity to account missing weights.

Parameters

- **n_total** (int) – total number of events in entire data set
- **wgt_name** (str) – name of weight group in fold file to use. N.B. if you have reweighted to balance classes, be sure to use the un-reweighted weights.
- **targ_name** (str) – name of target group in fold file which indicates whether the event is signal or background
- **zero_preds** (List[str]) – list of predicted classes which correspond to class 0 in the form pred_[i], where i is a NN output index
- **one_preds** (List[str]) – list of predicted classes which correspond to class 1 in the form pred_[i], where i is a NN output index
- **br** (float) – constant bias offset for background yield

- **syst_unc_b** (float) – fractional systematic uncertainty on background yield
- **use_quick_scan** (bool) – whether to optimise AMS by the *ams_scan_quick()* method (fast but suffers floating point precision) if False use *ams_scan_slow()* (slower but more accurate)
- **name** (Optional[str]) – optional name for metric, otherwise will be 'AMS'
- **main_metric** (bool) – whether this metic should be treated as the primary metric for SaveBest and EarlyStopping Will automatically set the first EvalMetric to be main if multiple primary metrics are submitted

Examples::

```
>>> ams_metric = MultiAMS(n_total=250000, br=10, targ_name='gen_target',
...                       wgt_name='gen_orig_weight',
...                       zero_preds=['pred_0', 'pred_1', 'pred_2'],
...                       one_preds=['pred_3'])
>>>
>>> ams_metric = MultiAMS(n_total=250000, syst_unc_b=0.1,
...                       targ_name='gen_target',
...                       wgt_name='gen_orig_weight',
...                       use_quick_scan=False,
...                       zero_preds=['pred_0', 'pred_1', 'pred_2'],
...                       one_preds=['pred_3'])
```

evaluate()

Compute maximum AMS on fold using provided predictions.

Return type float

Returns Maximum AMS computed on reweighted data from fold

class lumin.nn.metrics.class_eval.**BinaryAccuracy**(*threshold=0.5*, *name='Acc'*, *main_metric=True*)

Bases: *lumin.nn.metrics.eval_metric.EvalMetric*

Computes and returns the accuracy of a single-output model for binary classification tasks.

Parameters

- **threshold** (float) – minimum value of model prediction that will be considered a prediction of class 1. Values below this threshold will be considered predictions of class 0. Default = 0.5.
- **name** (Optional[str]) – optional name for metric, otherwise will be 'Acc'
- **main_metric** (bool) – whether this metic should be treated as the primary metric for SaveBest and EarlyStopping Will automatically set the first EvalMetric to be main if multiple primary metrics are submitted

Examples::

```
>>> acc_metric = BinaryAccuracy()
>>>
>>> acc_metric = BinaryAccuracy(threshold=0.8)
```

evaluate()

Computes the (weighted) accuracy for a set of targets and predictions for a given threshold.

Return type float

Returns The (weighted) accuracy for the specified threshold

class lumin.nn.metrics.class_eval.**RocAucScore**(*average='macro'*, *max_fpr=None*, *multi_class='raise'*, *name='ROC AUC'*, *main_metric=True*)

Bases: *lumin.nn.metrics.eval_metric.EvalMetric*

Computes and returns the area under the Receiver Operator Characteristic curve (ROC AUC) of a classifier model.

Parameters

- **average** (Optional[str]) – As per scikit-learn. {'micro', 'macro', 'samples', 'weighted'} or None, default='macro' If None, the scores for each class are returned. Otherwise, this determines the type of averaging performed on the data: Note: multiclass ROC AUC currently only handles the 'macro' and 'weighted' averages.

  'micro': Calculate metrics globally by considering each element of the label indicator matrix as a label.

  'macro': Calculate metrics for each label, and find their unweighted mean. This does not take label imbalance into account.

  'weighted': Calculate metrics for each label, and find their average, weighted by support (the number of true instances for each label).

  'samples': Calculate metrics for each instance, and find their average.

  Will be ignored when y_true is binary.

- **max_fpr** (Optional[float]) – As per scikit-learn. float > 0 and <= 1, default=None If not None, the standardized partial AUC over the range [0, max_fpr] is returned. For the multiclass case, max_fpr, should be either equal to None or 1.0 as AUC ROC partial computation currently is not supported for multiclass.

- **multi_class** (str) – As per scikit-learn. {'raise', 'ovr', 'ovo'}, default='raise' Multiclass only. Determines the type of configuration to use. The default value raises an error, so either 'ovr' or 'ovo' must be passed explicitly.

  'ovr': Computes the AUC of each class against the rest. This treats the multiclass case in the same way as the multilabel case. Sensitive to class imbalance even when average == 'macro', because class imbalance affects the composition of each of the 'rest' groupings.

  'ovo': Computes the average AUC of all possible pairwise combinations of classes. Insensitive to class imbalance when average == 'macro'.

- **name** (Optional[str]) – optional name for metric, otherwise will be 'Acc'

- **main_metric** (bool) – whether this metic should be treated as the primary metric for SaveBest and EarlyStopping Will automatically set the first EvalMetric to be main if multiple primary metrics are submitted

Examples::

```
>>> auc_metric = RocAucScore()
>>>
>>> auc_metric = RocAucScore(max_fpr=0.2)
>>>
>>> auc_metric = RocAucScore(multi_class='ovo')
```

evaluate()

Computes the (weighted) (averaged) ROC AUC for a set of targets and predictions.

Return type float

Returns The (weighted) (averaged) ROC AUC for the specified threshold

lumin.nn.metrics.eval_metric module

class lumin.nn.metrics.eval_metric.**EvalMetric**(*name*, *lower_metric_better*, *main_metric=True*)

Bases: *lumin.nn.callbacks.callback.Callback*

Abstract class for evaluating performance of a model using some metric

Parameters

- **name** (Optional[str]) – optional name for metric, otherwise will be inferred from class
- **lower_metric_better** (bool) – whether a lower metric value should be treated as representing better perofrmance
- **main_metric** (bool) – whether this metic should be treated as the primary metric for SaveBest and EarlyStopping Will automatically set the first EvalMetric to be main if multiple primary metrics are submitted

abstract evaluate()

Evaluate the required metric for a given fold and set of predictions

Return type float

Returns metric value

evaluate_model(*model*, *fy*, *fold_idx*, *inputs*, *targets*, *weights=None*, *bs=None*)

Gets model predicitons and computes metric value. fy and fold_idx arguments necessary in case the metric requires extra information beyond inputs, tragets, and weights.

Parameters

- **model** (AbsModel) – model to evaluate
- **fy** (*FoldYielder*) – *FoldYielder* containing data
- **fold_idx** (int) – fold index of corresponding data
- **inputs** (ndarray) – input data
- **targets** (ndarray) – target data
- **weights** (Optional[ndarray]) – optional weights
- **bs** (Optional[int]) – optional batch size

Return type float

Returns metric value

evaluate_preds(*fy*, *fold_idx*, *preds*, *targets*, *weights=None*)

Computes metric value from predictions. fy and fold_idx arguments necessary in case the metric requires extra information beyond inputs, tragets, and weights.

Parameters

- **fy** (*FoldYielder*) – *FoldYielder* containing data
- **fold_idx** (int) – fold index of corresponding data
- **inputs** – input data
- **targets** (ndarray) – target data

- **weights** (Optional[ndarray]) – optional weights
- **bs** – optional batch size

Return type float

Returns metric value

get_df()
Returns a DataFrame for the given fold containing targets, weights, and predictions

Return type DataFrame

Returns DataFrame for the given fold containing targets, weights, and predictions

get_metric()
Returns metric value

Return type float

Returns metric value

on_epoch_begin()
Resets prediction tracking

Return type None

on_epoch_end()
Compute metric using saved predictions

Return type None

on_forwards_end()
Save predictions from batch

Return type None

on_train_begin()
Ensures that only one main metric is used

Return type None

lumin.nn.metrics.reg_eval module

class lumin.nn.metrics.reg_eval.**RegPull**(*return_mean*, *use_bootstrap=False*, *use_pull=True*, *name=None*, *main_metric=True*)
Bases: *lumin.nn.metrics.eval_metric.EvalMetric*

Compute mean or standard deviation of delta or pull of some feature which is being directly regressed to. Optionally, use bootstrap resampling on validation data.

Parameters

- **return_mean** (bool) – whether to return the mean or the standard deviation
- **use_bootstrap** (bool) – whether to bootstrap resamples validation fold when computing statisitic
- **use_pull** (bool) – whether to return the pull (differences / targets) or delta (differences)
- **name** (Optional[str]) – optional name for metric, otherwise will be inferred from *use_pull*

- **main_metric** (bool) – whether this metic should be treated as the primary metric for SaveBest and EarlyStopping Will automatically set the first EvalMetric to be main if multiple primary metrics are submitted

Examples::

```
>>> mean_pull  = RegPull(return_mean=True, use_bootstrap=True,
...                      use_pull=True)
>>>
>>> std_delta  = RegPull(return_mean=False, use_bootstrap=True,
...                      use_pull=False)
>>>
>>> mean_pull  = RegPull(return_mean=True, use_bootstrap=False,
...                      use_pull=True, wgt_name='weights')
```

evaluate()

Compute mean or width of regression error.

Return type float

Returns Mean or width of regression error

class lumin.nn.metrics.reg_eval.**RegAsProxyPull**(*proxy_func*, *return_mean*, *targ_name=None*, *use_bootstrap=False*, *use_pull=True*, *name=None*, *main_metric=True*)

Bases: *lumin.nn.metrics.reg_eval.RegPull*

Compute mean or standard deviation of delta or pull of some feature which is being indirectly regressed to via a proxy function. Optionally, use bootstrap resampling on validation data.

Parameters

- **proxy_func** (Callable[[DataFrame], None]) – function which acts on regression predictions and adds pred and gen_target columns to the Pandas DataFrame it is passed which contains prediction columns pred_{i}
- **return_mean** (bool) – whether to return the mean or the standard deviation
- **use_bootstrap** (bool) – whether to bootstrap resamples validation fold when computing statisitic
- **use_weights** – whether to actually use weights if wgt_name is set
- **use_pull** (bool) – whether to return the pull (differences / targets) or delta (differences)
- **targ_name** (Optional[str]) – optional name of group in fold file containing regression targets
- **name** (Optional[str]) – optional name for metric, otherwise will be inferred from *use_pull*
- **main_metric** (bool) – whether this metic should be treated as the primary metric for SaveBest and EarlyStopping Will automatically set the first EvalMetric to be main if multiple primary metrics are submitted

Examples::

```
>>> def reg_proxy_func(df):
>>>     df['pred'] = calc_pair_mass(df, (1.77682, 1.77682),
...                                 {targ[targ.find('_t')+3:]:
...                                 f'pred_{i}' for i, targ
```

(continues on next page)

(continued from previous page)

```
...                                     in enumerate(targ_feats)})
>>>     df['gen_target'] = 125
>>>
>>> std_delta = RegAsProxyPull(proxy_func=reg_proxy_func,
...                            return_mean=False, use_pull=False)
```

evaluate()

Compute statisitic on fold using provided predictions.

Parameters

- **fy** – *FoldYielder* interfacing to data
- **idx** – fold index corresponding to fold for which y_pred was computed
- **y_pred** – predictions for fold

Return type float

Returns Statistic set in initialisation computed on the chsoen fold

Examples::

```
>>> mean = mean_pull.evaluate(train_fy, val_id, val_preds)
```

Module contents

6.1.7 lumin.nn.models package

Subpackages

lumin.nn.models.blocks package

Submodules

lumin.nn.models.blocks.body module

class lumin.nn.models.blocks.body.**IdentBody**(*n_in*, *feat_map*, *lookup_init=<function lookup_normal_init>*, *lookup_act=<function lookup_act>*, *freeze=False*, *bn_class=<class 'torch.nn.modules.batchnorm.BatchNorm1d'>*)

Bases: lumin.nn.models.blocks.body.AbsBody

Placeholder body module for cases in which a body is not required. Outputs are equal to imputs.

forward(*x*)

Pass tensor through block

Parameters **x** (Tensor) – input tensor

Returns Resulting tensor

Return type Tensor

get_out_size()
Get size width of output layer

Return type int

Returns Width of output layer

class lumin.nn.models.blocks.body.**FullyConnected**(*n_in*, *feat_map*, *depth*, *width*, *do=0*, *bn=False*, *act='relu'*, *res=False*, *dense=False*, *growth_rate=0*, *lookup_init=<function lookup_normal_init>*, *lookup_act=<function lookup_act>*, *freeze=False*, *bn_class=<class 'torch.nn.modules.batchnorm.BatchNorm1d'>*)

Bases: lumin.nn.models.blocks.body.AbsBody

Fully connected set of hidden layers. Designed to be passed as a 'body' to *ModelBuilder*. Supports batch normalisation and dropout. Order is dense->activation->BN->DO, except when res is true in which case the BN is applied after the addition. Can optionaly have skip connections between each layer (res=true). Alternatively can concatinate layers (dense=true) growth_rate parameter can be used to adjust the width of layers according to width+(width*(depth-1)*growth_rate)

Parameters

- **n_in** (int) – number of inputs to the block
- **feat_map** (Dict[str, List[int]]) – dictionary mapping input features to the model to outputs of head block
- **depth** (int) – number of hidden layers. If res==True and depth is even, depth will be increased by one.
- **width** (int) – base width of each hidden layer
- **do** (float) – if not None will add dropout layers with dropout rates do
- **bn** (bool) – whether to use batch normalisation
- **act** (str) – string representation of argument to pass to lookup_act
- **res** (bool) – whether to add an additative skip connection every two dense layers. Mutually exclusive with dense.
- **dense** (bool) – whether to perform layer-wise concatinations after every layer. Mutually exclusion with res.
- **growth_rate** (int) – rate at which width of dense layers should increase with depth beyond the initial layer. Ignored if res=True. Can be negative.
- **lookup_init** (Callable[[str, Optional[int], Optional[int]], Callable[[Tensor], None]]) – function taking choice of activation function, number of inputs, and number of outputs an returning a function to initialise layer weights.
- **lookup_act** (Callable[[str], Any]) – function taking choice of activation function and returning an activation function layer
- **freeze** (bool) – whether to start with module parameters set to untrainable
- **bn_class** (Callable[[int], Module]) – class to use for BatchNorm, default is *nn.BatchNorm1d*

Examples::

```
>>> body = FullyConnected(n_in=32, feat_map=head.feat_map, depth=4,
...                       width=100, act='relu')
>>>
>>> body = FullyConnected(n_in=32, feat_map=head.feat_map, depth=4,
...                       width=200, act='relu', growth_rate=-0.3)
>>>
>>> body = FullyConnected(n_in=32, feat_map=head.feat_map, depth=4,
...                       width=100, act='swish', do=0.1, res=True)
>>>
>>> body = FullyConnected(n_in=32, feat_map=head.feat_map, depth=6,
...                       width=32, act='selu', dense=True,
...                       growth_rate=0.5)
>>>
>>> body = FullyConnected(n_in=32, feat_map=head.feat_map, depth=6,
...                       width=50, act='prelu', bn=True,
...                       lookup_init=lookup_uniform_init)
```

forward(*x*)

Pass tensor through block

Parameters **x** (Tensor) – input tensor

Returns Resulting tensor

Return type Tensor

get_out_size()

Get size width of output layer

Return type int

Returns Width of output layer

class lumin.nn.models.blocks.body.**MultiBlock**(*n_in*, *feat_map*, *blocks*, *feats_per_block*, *bottleneck_sz=0*, *bottleneck_act=None*, *lookup_init=<function lookup_normal_init>*, *lookup_act=<function lookup_act>*, *freeze=False*)

Bases: lumin.nn.models.blocks.body.AbsBody

Body block allowing outputs of head block to be split amongst a series of body blocks. Output is the concatination of all sub-body blocks. Optionally, single-neuron 'bottleneck' layers can be used to pass an input to each sub-block based on a learned function of the input features that block would otherwise not receive, i.e. a highly compressed representation of the rest of teh feature space.

Parameters

- **n_in** (int) – number of inputs to the block
- **feat_map** (Dict[str, List[int]]) – dictionary mapping input features to the model to outputs of head block
- **blocks** (List[partial]) – list of uninstantciated AbsBody blocks to which to pass a subsection of the total inputs. Note that partials should be used to set any relevant parameters at initialisation time
- **feats_per_block** (List[List[str]]) – list of lists of names of features to pass to each AbsBody, not that the feat_map provided by AbsHead will map features to their relavant head outputs

- **bottleneck** – if true, each block will receive the output of a single neuron which takes as input all the features which each given block does not directly take as inputs
- **bottleneck_act** (Optional[str]) – if set to a string representation of an activation function, the output of each bottleneck neuron will be passed throguh the defined activation function before being passed to their associated blocks
- **lookup_init** (Callable[[str, Optional[int], Optional[int]], Callable[[Tensor], None]]) – function taking choice of activation function, number of inputs, and number of outputs an returning a function to initialise layer weights.
- **lookup_act** (Callable[[str], Any]) – function taking choice of activation function and returning an activation function layer
- **freeze** (bool) – whether to start with module parameters set to untrainable

Examples::

```
>>> body = MultiBlock(
...     blocks=[partial(FullyConnected, depth=1, width=50, act='swish'),
...             partial(FullyConnected, depth=6, width=55, act='swish',
...                     dense=True, growth_rate=-0.1)],
...     feats_per_block=[[f for f in train_feats if 'DER_' in f],
...                      [f for f in train_feats if 'PRI_' in f]])
>>>
>>> body = MultiBlock(
...     blocks=[partial(FullyConnected, depth=1, width=50, act='swish'),
...     partial(FullyConnected, depth=6, width=55, act='swish',
...             dense=True, growth_rate=-0.1)],
...     feats_per_block=[[f for f in train_feats if 'DER_' in f],
...                      [f for f in train_feats if 'PRI_' in f]],
...     bottleneck=True)
>>>
>>> body = MultiBlock(
...     blocks=[partial(FullyConnected, depth=1, width=50, act='swish'),
...             partial(FullyConnected, depth=6, width=55, act='swish',
...                     dense=True, growth_rate=-0.1)],
...     feats_per_block=[[f for f in train_feats if 'DER_' in f],
...                      [f for f in train_feats if 'PRI_' in f]],
...     bottleneck=True, bottleneck_act='swish')
```

forward(*x*)

Pass tensor through block

Parameters **x** (Tensor) – input tensor

Returns Resulting tensor

Return type Tensor

get_out_size()

Get size width of output layer

Return type int

Returns Total number of outputs across all blocks

lumin.nn.models.blocks.conv_blocks module

class lumin.nn.models.blocks.conv_blocks.**Conv1DBlock**(*in_c*, *out_c*, *kernel_sz*, *padding='auto'*, *stride=1*, *act='relu'*, *bn=False*, *lookup_init=<function lookup_normal_init>*, *lookup_act=<function lookup_act>*, *bn_class=<class 'torch.nn.modules.batchnorm.BatchNorm1d'>*)

Bases: `torch.nn.modules.module.Module`

Basic building block for a building and applying a single 1D convolutional layer.

Parameters

- **in_c** (`int`) – number of input channels (number of features per object / rows in input matrix)
- **out_c** (`int`) – number of output channels (number of features / rows in output matrix)
- **kernel_sz** (`int`) – width of kernel, i.e. the number of columns to overlay
- **padding** (`Union[int, str]`) – amount of padding columns to add at start and end of convolution. If left as 'auto', padding will be automatically computed to conserve the number of columns.
- **stride** (`int`) – number of columns to move kernel when computing convolutions. Stride 1 = kernel centred on each column, stride 2 = kernel centred on ever other column and input size halved, et cetera.
- **act** (`str`) – string representation of argument to pass to lookup_act
- **bn** (`bool`) – whether to use batch normalisation (default order weights->activation->batchnorm)
- **lookup_init** (`Callable[[str, Optional[int], Optional[int]], Callable[[Tensor], None]]`) – function taking choice of activation function, number of inputs, and number of outputs an returning a function to initialise layer weights.
- **lookup_act** (`Callable[[str], Any]`) – function taking choice of activation function and returning an activation function layer
- **bn_class** (`Callable[[int], Module]`) – class to use for BatchNorm, default is *nn.BatchNorm1d*

Examples::

```
>>> conv = Conv1DBlock(in_c=3, out_c=16, kernel_sz=3)
>>>
>>> conv = Conv1DBlock(in_c=16, out_c=32, kernel_sz=3, stride=2)
>>>
>>> conv = Conv1DBlock(in_c=3, out_c=16, kernel_sz=3, act='swish', bn=True)
```

forward(*x*)

Passes input through the layers. Might need to be overloaded in inheritance, depending on architecture.

Parameters **x** (`Tensor`) – input tensor

Return type `Tensor`

Returns Resulting tensor

get_conv_layer(*in_c*, *out_c*, *kernel_sz*, *padding='auto'*, *stride=1*, *pre_act=False*, *groups=1*)
Builds a sandwich of layers with a single concilutional layer, plus any requested batch norm and activation. Also initialises layers to requested scheme.

Parameters

- **in_c** (int) – number of input channels (number of features per object / rows in input matrix)
- **out_c** (int) – number of output channels (number of features / rows in output matrix)
- **kernel_sz** (int) – width of kernel, i.e. the number of columns to overlay
- **padding** (Union[int, str]) – amount of padding columns to add at start and end of convolution. If left as 'auto', padding will be automatically computed to conserve the number of columns.
- **stride** (int) – number of columns to move kernel when computing convolutions. Stride 1 = kernel centred on each column, stride 2 = kernel centred on ever other column and input size halved, et cetera.
- **pre_act** (bool) – whether to apply batchnorm and activation layers prior to the weight layer, or afterwards
- **groups** (int) – number of blocks of connections from input channels to output channels

Return type Module

static get_padding(*kernel_sz*)
Automatically computes the required padding to keep the number of columns equal before and after convolution

Parameters **kernel_sz** (int) – width of convolutional kernel

Return type int

Returns size of padding

set_layers()
One of the main function to overload when inheriting from class. By default calls *self.get_conv_layer* once but can be changed to produce more complicated architectures. Sets *self.layers* to the constructed architecture.

Return type None

class lumin.nn.models.blocks.conv_blocks.**Res1DBlock**(*in_c*, *out_c*, *kernel_sz*, *padding='auto'*, *stride=1*, *act='relu'*, *bn=False*, *lookup_init=<function lookup_normal_init>*, *lookup_act=<function lookup_act>*, *bn_class=<class 'torch.nn.modules.batchnorm.BatchNorm1d'>*)
Bases: *lumin.nn.models.blocks.conv_blocks.Conv1DBlock*

Basic building block for a building and applying a pair of residually connected 1D convolutional layers (https://arxiv.org/abs/1512.03385). Batchnorm is applied 'pre-activation' as per https://arxiv.org/pdf/1603.05027.pdf, and convolutional shortcuts (again https://arxiv.org/pdf/1603.05027.pdf) are used when the stride of the first layer is greater than 1, or the number of input channels does not equal the number of output channels.

Parameters

- **in_c** (int) – number of input channels (number of features per object / rows in input matrix)
- **out_c** (int) – number of output channels (number of features / rows in output matrix)
- **kernel_sz** (int) – width of kernel, i.e. the number of columns to overlay
- **padding** (Union[int, str]) – amount of padding columns to add at start and end of convolution. If left as 'auto', padding will be automatically computed to conserve the number of columns.
- **stride** (int) – number of columns to move kernel when computing convolutions. Stride 1 = kernel centred on each column, stride 2 = kernel centred on ever other column and input size halved, et cetera.
- **act** (str) – string representation of argument to pass to lookup_act
- **bn** (bool) – whether to use batch normalisation (order is pre-activation: batchnorm->activation->weights)
- **lookup_init** (Callable[[str, Optional[int], Optional[int]], Callable[[Tensor], None]]) – function taking choice of activation function, number of inputs, and number of outputs an returning a function to initialise layer weights.
- **lookup_act** (Callable[[str], Any]) – function taking choice of activation function and returning an activation function layer

Examples::

```
>>> conv = Res1DBlock(in_c=16, out_c=16, kernel_sz=3)
>>>
>>> conv = Res1DBlock(in_c=16, out_c=32, kernel_sz=3, stride=2)
>>>
>>> conv = Res1DBlock(in_c=16, out_c=16, kernel_sz=3, act='swish', bn=True)
```

forward(*x*)

Passes input through the pair of layers and then adds the resulting tensor to the original input, which may be passed through a shortcut connection is necessary.

Parameters **x** (Tensor) – input tensor

Return type Tensor

Returns Resulting tensor

set_layers()

Constructs a pair of pre-activation convolutional layers, and a shortcut layer if necessary.

class lumin.nn.models.blocks.conv_blocks.**ResNeXt1DBlock**(*in_c*, *inter_c*, *cardinality*, *out_c*, *kernel_sz*, *padding='auto'*, *stride=1*, *act='relu'*, *bn=False*, *lookup_init=<function lookup_normal_init>*, *lookup_act=<function lookup_act>*, *bn_class=<class 'torch.nn.modules.batchnorm.BatchNorm1d'>*)

Bases: *lumin.nn.models.blocks.conv_blocks.Conv1DBlock*

Basic building block for a building and applying a set of residually connected groups of 1D convolutional layers (https://arxiv.org/abs/1611.05431). Batchnorm is applied 'pre-activation' as per https://arxiv.org/pdf/1603.05027.pdf, and convolutional shortcuts (again https://arxiv.org/pdf/1603.05027.pdf) are used when the stride of the first layer is greater than 1, or the number of input channels does not equal the number of output channels.

Parameters

- **in_c** (int) – number of input channels (number of features per object / rows in input matrix)
- **inter_c** (int) – number of intermediate channels in groups
- **cardinality** (int) – number of groups
- **out_c** (int) – number of output channels (number of features / rows in output matrix)
- **kernel_sz** (int) – width of kernel, i.e. the number of columns to overlay
- **padding** (Union[int, str]) – amount of padding columns to add at start and end of convolution. If left as 'auto', padding will be automatically computed to conserve the number of columns.
- **stride** (int) – number of columns to move kernel when computing convolutions. Stride 1 = kernel centred on each column, stride 2 = kernel centred on ever other column and input size halved, et cetera.
- **act** (str) – string representation of argument to pass to lookup_act
- **bn** (bool) – whether to use batch normalisation (order is pre-activation: batchnorm->activation->weights)
- **lookup_init** (Callable[[str, Optional[int], Optional[int]], Callable[[Tensor], None]]) – function taking choice of activation function, number of inputs, and number of outputs an returning a function to initialise layer weights.
- **lookup_act** (Callable[[str], Any]) – function taking choice of activation function and returning an activation function layer
- **bn_class** (Callable[[int], Module]) – class to use for BatchNorm, default is *nn.BatchNorm1d*

Examples::

```
>>> conv = ResNeXt1DBlock(in_c=32, inter_c=4, cardinality=4, out_c=32,␣
↪kernel_sz=3)
>>>
>>> conv = ResNeXt1DBlock(in_c=32, inter_c=4, cardinality=4, out_c=32,␣
↪kernel_sz=3, stride=2)
>>>
>>> conv = ResNeXt1DBlock(in_c=32, inter_c=4, cardinality=4, out_c=32,␣
↪kernel_sz=3, act='swish', bn=True)
```

forward(*x*)

Passes input through the set of layers and then adds the resulting tensor to the original input, which may be passed through a shortcut connection is necessary.

Parameters **x** (Tensor) – input tensor

Return type Tensor

Returns Resulting tensor

set_layers()
Constructs a set of grouped pre-activation convolutional layers, and a shortcut layer if necessary.

class lumin.nn.models.blocks.conv_blocks.**AdaptiveAvgMaxConcatPool1d**(*sz=None*)
Bases: torch.nn.modules.module.Module

Layer that reduces the size of each channel to the specified size, via two methods: average pooling and max pooling. The outputs are then concatenated channelwise.

Parameters **sz** (Union[int, Tuple[int, ...], None]) – Requested output size, default reduces each channel to 2*1 elements. The first element is the maximum value in the channel and the other is the average value in the channel.

forward(*x*)
Passes input through the adaptive pooling.

Parameters **x** – input tensor

Returns Resulting tensor

class lumin.nn.models.blocks.conv_blocks.**AdaptiveAvgMaxConcatPool2d**(*sz=None*)
Bases: *lumin.nn.models.blocks.conv_blocks.AdaptiveAvgMaxConcatPool1d*

Layer that reduces the size of each channel to the specified size, via two methods: average pooling and max pooling. The outputs are then concatenated channelwise.

Parameters **sz** (Union[int, Tuple[int, ...], None]) – Requested output size, default reduces each channel to 2*1 elements. The first element is the maximum value in the channel and the other is the average value in the channel.

class lumin.nn.models.blocks.conv_blocks.**AdaptiveAvgMaxConcatPool3d**(*sz=None*)
Bases: *lumin.nn.models.blocks.conv_blocks.AdaptiveAvgMaxConcatPool1d*

Layer that reduces the size of each channel to the specified size, via two methods: average pooling and max pooling. The outputs are then concatenated channelwise.

Parameters **sz** (Union[int, Tuple[int, ...], None]) – Requested output size, default reduces each channel to 2*1 elements. The first element is the maximum value in the channel and the other is the average value in the channel.

class lumin.nn.models.blocks.conv_blocks.**SEBlock1d**(*n_in*, *r*, *act='relu'*, *lookup_init=<function lookup_normal_init>*, *lookup_act=<function lookup_act>*)
Bases: torch.nn.modules.module.Module

Squeeze-excitation block [Hu, Shen, Albanie, Sun, & Wu, 2017](https://arxiv.org/abs/1709.01507). Incoming data is averaged per channel, fed through a single layer of width *n_in//r* and the chose activation, then a second layer of width *n_in* and a sigmoid activation. Channels in the original data are then multiplied by the learned channe weights.

Parameters

- **n_in** (int) – number of incoming channels
- **r** (int) – the reduction ratio for the channel compression
- **act** (str) – string representation of argument to pass to lookup_act
- **lookup_init** (Callable[[str, Optional[int], Optional[int]], Callable[[Tensor], None]]) – function taking choice of activation function, number of inputs, and number of outputs an returning a function to initialise layer weights.
- **lookup_act** (Callable[[str], Any]) – function taking choice of activation function and returning an activation function layer

forward(*x*)

Rescale the incoming tensor by the learned channel weights

Parameters **x** (Tensor) – incoming tensor

Return type Tensor

Returns x*y, where y is the output of the squeeze-excitation network

class lumin.nn.models.blocks.conv_blocks.**SEBlock2d**(*n_in*, *r*, *act='relu'*, *lookup_init=<function lookup_normal_init>*, *lookup_act=<function lookup_act>*)

Bases: *lumin.nn.models.blocks.conv_blocks.SEBlock1d*

Squeeze-excitation block [Hu, Shen, Albanie, Sun, & Wu, 2017](https://arxiv.org/abs/1709.01507). Incoming data is averaged per channel, fed through a single layer of width *n_in//r* and the chose activation, then a second layer of width *n_in* and a sigmoid activation. Channels in the original data are then multiplied by the learned channe weights.

Parameters

- **n_in** (int) – number of incoming channels
- **r** (int) – the reduction ratio for the channel compression
- **act** (str) – string representation of argument to pass to lookup_act
- **lookup_init** (Callable[[str, Optional[int], Optional[int]], Callable[[Tensor], None]]) – function taking choice of activation function, number of inputs, and number of outputs an returning a function to initialise layer weights.
- **lookup_act** (Callable[[str], Any]) – function taking choice of activation function and returning an activation function layer

class lumin.nn.models.blocks.conv_blocks.**SEBlock3d**(*n_in*, *r*, *act='relu'*, *lookup_init=<function lookup_normal_init>*, *lookup_act=<function lookup_act>*)

Bases: *lumin.nn.models.blocks.conv_blocks.SEBlock1d*

Squeeze-excitation block [Hu, Shen, Albanie, Sun, & Wu, 2017](https://arxiv.org/abs/1709.01507). Incoming data is averaged per channel, fed through a single layer of width *n_in//r* and the chose activation, then a second layer of width *n_in* and a sigmoid activation. Channels in the original data are then multiplied by the learned channe weights.

Parameters

- **n_in** (int) – number of incoming channels
- **r** (int) – the reduction ratio for the channel compression
- **act** (str) – string representation of argument to pass to lookup_act
- **lookup_init** (Callable[[str, Optional[int], Optional[int]], Callable[[Tensor], None]]) – function taking choice of activation function, number of inputs, and number of outputs an returning a function to initialise layer weights.
- **lookup_act** (Callable[[str], Any]) – function taking choice of activation function and returning an activation function layer

lumin.nn.models.blocks.endcap module

class lumin.nn.models.blocks.endcap.**AbsEndcap**(*model*)
Bases: torch.nn.modules.module.Module

Abstract class for constructing post training layer which performs further calculation on NN outputs. Used when NN was trained to some proxy objective

Parameters **model** (Module) – trained *Model* to wrap

forward(*x*)
Pass tensor through endcap and compute function

Parameters **x** (Tensor) – model output tensor

Returns Resulting tensor

Return type Tensor

abstract func(*x*)
Transformation functio to apply to model outputs

Arguements: x: model output tensor

Return type Tensor

Returns Resulting tensor

predict(*inputs*, *as_np=True*)
Evaluate model on input tensor, and comput function of model outputs

Parameters

- **inputs** (Union[ndarray, DataFrame, Tensor]) – input data as Numpy array, Pandas DataFrame, or tensor on device
- **as_np** (bool) – whether to return predictions as Numpy array (otherwise tensor)

Return type Union[ndarray, Tensor]

Returns model predictions pass through endcap function

lumin.nn.models.blocks.gnn_blocks module

class lumin.nn.models.blocks.gnn_blocks.**GraphCollapser**(*n_v*, *n_fpv*, *flatten*, *f_initial_outs=None*, *n_sa_layers=0*, *sa_width=None*, *f_final_outs=None*, *global_feat_vec=False*, *agg_methods=['mean', 'max']*, *do=0*, *bn=False*, *act='relu'*, *lookup_init=<function lookup_normal_init>*, *lookup_act=<function lookup_act>*, *bn_class=<class 'torch.nn.modules.batchnorm.BatchNorm1d'>*, *sa_class=<class 'lumin.nn.models.layers.self_attention.SelfAttention'>*)

Bases: lumin.nn.models.blocks.gnn_blocks.AbsGraphBlock

Class for collapsing features per vertex (batch x vertices x features) down to flat data (batch x features). Can act in two ways:

1. Compute aggregate features by taking the average and maximum of each feature across all vertices (does not assume any order to the vertices)
2. Flatten out the vertices by reshaping (does assume an ordering to the vertices)

Regardless of flattening approach, features per vertex can be revised beforehand via neural networks and self-attention.

Parameters

- **n_v** (int) – number of vertices per data point to expect
- **n_fpv** (int) – number of features per vertex to expect
- **flatten** (bool) – if True will flatten (reshape) data into (batch x features), otherwise will compute aggregate features (average and max)
- **f_initial_outs** (Optional[List[int]]) – list of widths for the NN layers in an NN before self-attention (None = no NN)
- **n_sa_layers** (int) – number of self-attention layers (outputs will be fed into subsequent layers)
- **sa_width** (Optional[int]) – width of self attention representation (paper recommends n_fpv//4)
- **f_final_outs** (Optional[List[int]]) – list of widths for the NN layers in an NN after self-attention (None = no NN)
- **global_feat_vec** (bool) – if true and f_initial_outs or f_final_outs are not None, will concatenate the mean of each feature as new features to each vertex prior to the last network.
- **agg_methods** (Union[List[str], str]) – list of text representations of aggregation methods. Default is mean and max.
- **do** (float) – dropout rate to be applied to hidden layers in the NNs
- **bn** (bool) – whether batch normalisation should be applied to hidden layers in the NNs

- **act** (str) – activation function to apply to hidden layers in the NNs
- **lookup_init** (Callable[[str, Optional[int], Optional[int]], Callable[[Tensor], None]]) – function taking choice of activation function, number of inputs, and number of outputs an returning a function to initialise layer weights.
- **lookup_act** (Callable[[str], Any]) – function taking choice of activation function and returning an activation function layer
- **bn_class** (Callable[[int], Module]) – class to use for BatchNorm, default is *LCBatchNorm1d*
- **sa_class** (Callable[[int], Module]) – class to use for self-attention layers, default is *SelfAttention*

forward(*x*)

Collapses features per vertex down to features

Arguemnts: x: incoming data (batch x vertices x features)

Return type Tensor

Returns Flattened data (batch x flat features)

get_out_size()

Get size of output

Return type int

Returns Width of output representation

class lumin.nn.models.blocks.gnn_blocks.**NodePredictor**(*n_v*, *n_fpv*, *out_act*, *transpose_out*, *f_initial_outs=None*, *n_sa_layers=0*, *sa_width=None*, *f_final_outs=None*, *global_feat_vec=False*, *do=0*, *bn=False*, *act='relu'*, *lookup_init=<function lookup_normal_init>*, *lookup_act=<function lookup_act>*, *bn_class=<class 'torch.nn.modules.batchnorm.BatchNorm1d'>*, *sa_class=<class 'lumin.nn.models.layers.self_attention.SelfAttention'>*)

Bases: *lumin.nn.models.blocks.gnn_blocks.GraphCollapser*

Modified *GraphCollapser* for providing a set of predictions per node in a graph, i.e collapsing features per vertex (batch x vertices x features) down to predictions per vertex, with approriate output activation functions data (batch x vertices x predictions). For compatibility with the format expected by some loss functions (e.g. *torch.nn.NLLLoss*), the output can be transposed to (batch x predictions x vertices). Features per vertex can be revised beforehand via neural networks and self-attention. The *f_final* neural network can be used to transform the input size to the required number of predicitons per node.

Important: Since predictions are being provided in the *head* part of the model, but LUMIN expects models to also have a *body* and *tail* section. It is strongly recommended to use the *IdentBody* and *IdentTail*

modules to be placeholders.

Parameters

- **n_v** (int) – number of vertices per data point to expect
- **n_fpv** (int) – number of features per vertex to expect
- **out_act** (str) – Output activation function to apply to every set of prediction per vertex. The weight initialisation of the last NN layer will be automatically set.
- **transpose_out** (bool) – If True, will transpose the putput into (batch x predictions x vertices), otherwise the output will be (batch x vertices x predictions)
- **f_initial_outs** (Optional[List[int]]) – list of widths for the NN layers in an NN before self-attention (None = no NN)
- **n_sa_layers** (int) – number of self-attention layers (outputs will be fed into subsequent layers)
- **sa_width** (Optional[int]) – width of self attention representation (paper recommends n_fpv//4)
- **f_final_outs** (Optional[List[int]]) – list of widths for the NN layers in an NN after self-attention (None = no NN)
- **global_feat_vec** (bool) – if true and f_initial_outs or f_final_outs are not None, will concatenate the mean of each feature as new features to each vertex prior to the last network.
- **do** (float) – dropout rate to be applied to hidden layers in the NNs
- **bn** (bool) – whether batch normalisation should be applied to hidden layers in the NNs
- **act** (str) – activation function to apply to hidden layers in the NNs
- **lookup_init** (Callable[[str, Optional[int], Optional[int]], Callable[[Tensor], None]]) – function taking choice of activation function, number of inputs, and number of outputs an returning a function to initialise layer weights.
- **lookup_act** (Callable[[str], Any]) – function taking choice of activation function and returning an activation function layer
- **bn_class** (Callable[[int], Module]) – class to use for BatchNorm, default is *LCBatchNorm1d*
- **sa_class** (Callable[[int], Module]) – class to use for self-attention layers, default is *SelfAttention*

class lumin.nn.models.blocks.gnn_blocks.**InteractionNet**(*n_v*, *n_fpv*, *intfunc_outs*, *outfunc_outs*, *do=0*, *bn=False*, *act='relu'*, *lookup_init=<function lookup_normal_init>*, *lookup_act=<function lookup_act>*, *bn_class=<class 'torch.nn.modules.batchnorm.BatchNorm1d'>*)

Bases: lumin.nn.models.blocks.gnn_blocks.AbsGraphFeatExtractor

Implementation of the Interaction Graph-Network (https://arxiv.org/abs/1612.00222). Shown to be applicable for embedding many 4-momenta in e.g. https://arxiv.org/abs/1908.05318

Receives column-wise data and returns column-wise

Parameters

- **n_v** (int) – Number of vertices to expect per datapoint
- **n_fpv** (int) – number features per vertex
- **intfunc_outs** (List[int]) – list of widths for the internal NN layers
- **outfunc_outs** (List[int]) – list of widths for the output NN layers
- **do** (float) – dropout rate to be applied to hidden layers in the interaction-representation and post-interaction networks
- **bn** (bool) – whether batch normalisation should be applied to hidden layers in the interaction-representation and post-interaction networks
- **act** (str) – activation function to apply to hidden layers in the interaction-representation and post-interaction networks
- **lookup_init** (Callable[[str, Optional[int], Optional[int]], Callable[[Tensor], None]]) – function taking choice of activation function, number of inputs, and number of outputs an returning a function to initialise layer weights.
- **lookup_act** (Callable[[str], Any]) – function taking choice of activation function and returning an activation function layer
- **bn_class** (Callable[[int], Module]) – class to use for BatchNorm, default is *nn.BatchNorm1d*

Examples::

```
>>> inet = InteractionNet(n_v=128, n_fpv=10, intfunc_outs=[20,10], outfunc_
↪outs=[20,4])
```

forward(*x*)

Learn new features per vertex

Parameters **x** (Tensor) – columnwise matrix data (batch x features x vertices)

Return type Tensor

Returns columnwise matrix data (batch x new features x vertices)

get_out_size()

Return type Tuple[int, int]

row_wise = False

class lumin.nn.models.blocks.gnn_blocks.**GravNet**(*n_v*, *n_fpv*, *cat_means*, *f_slr_depth*, *n_s*, *n_lr*, *k*, *f_out_depth*, *n_out*, *gn_class=<class 'lumin.nn.models.blocks.gnn_blocks.GravNetLayer'>*, *use_sa=False*, *do=0*, *bn=False*, *act='relu'*, *lookup_init=<function lookup_normal_init>*, *lookup_act=<function lookup_act>*, *bn_class=<class 'torch.nn.modules.batchnorm.BatchNorm1d'>*, *sa_class=<class 'lumin.nn.models.layers.self_attention.SelfAttention'>*, ***kargs*)

Bases: lumin.nn.models.blocks.gnn_blocks.AbsGraphFeatExtractor

GravNet GNN head (Qasim, Kieseler, Iiyama, & Pierini, 2019 https://link.springer.com/article/10.1140/epjc/s10052-019-7113-9). Passes features per vertex (batch x vertices x features) through several GravNetLayer layers. Like the paper model, this has the option of caching and concatenating the outputs of each GravNet layer prior to the final layer. The features per vertex are then flattened/aggregated across the vertices to flat data (batch x features).

Parameters

- **n_v** (int) – Number of vertices to expect per datapoint
- **n_fpv** (int) – number features per vertex
- **cat_means** (bool) – if True, will extend the incoming features per vertex by including the means of all features across all vertices
- **f_slr_depth** (int) – number of layers to use for the latent rep. NN
- **n_s** (int) – number of latent-spatial dimensions to compute
- **n_lr** (int) – number of features to compute per vertex for latent representation
- **k** (int) – number of neighbours (including self) each vertex should consider when aggregating latent-representation features
- **f_out_depth** (int) – number of layers to use for the output NN
- **n_out** (Union[List[int], int]) – number of output features to compute per vertex, if a list will add multiple gravnet layers, each of which outputs the respective number of features
- **gn_class** (Callable[[Dict[str, Any]], *GravNetLayer*]) – class to use for GravNet layers, default is *GravNetLayer*
- **use_sa** (bool) – if true, will apply self-attention layer to the neighbourhhood features per vertex prior to aggregation
- **do** (float) – dropout rate to be applied to hidden layers in the NNs
- **bn** (bool) – whether batch normalisation should be applied to hidden layers in the NNs
- **act** (str) – activation function to apply to hidden layers in the NNs
- **lookup_init** (Callable[[str, Optional[int], Optional[int]], Callable[[Tensor], None]]) – function taking choice of activation function, number of inputs, and number of outputs an returning a function to initialise layer weights.

- **lookup_act** (Callable[[str], Any]) – function taking choice of activation function and returning an activation function layer
- **freeze** – whether to start with module parameters set to untrainable
- **bn_class** (Callable[[int], Module]) – class to use for BatchNorm, default is *nn.BatchNorm1d*
- **sa_class** (Callable[[int], Module]) – class to use for self-attention layers, default is *SelfAttention*

forward(*x*)

Passes input through the GravNet head.

Parameters **x** (Tensor) – row-wise tensor (batch x vertices x features)

Return type Tensor

Returns Resulting tensor row-wise tensor (batch x vertices x new features)

get_out_size()

Return type Tuple[int, int]

row_wise = True

class lumin.nn.models.blocks.gnn_blocks.**GravNetLayer**(*n_fpv*, *n_s*, *n_lr*, *k*, *agg_methods*, *n_out*, *cat_means=True*, *f_slr_depth=1*, *f_out_depth=1*, *potential=<function GravNetLayer.<lambda>>*, *use_sa=False*, *do=0*, *bn=False*, *act='relu'*, *lookup_init=<function lookup_normal_init>*, *lookup_act=<function lookup_act>*, *bn_class=<class 'torch.nn.modules.batchnorm.BatchNorm1d'>*, *sa_class=<class 'lumin.nn.models.layers.self_attention.SelfAttention'>*)

Bases: lumin.nn.models.blocks.gnn_blocks.AbsGraphBlock

Single GravNet GNN layer (Qasim, Kieseler, Iiyama, & Pierini, 2019 https://link.springer.com/article/10.1140/epjc/s10052-019-7113-9). Designed to be used as a sub-layer of a head block, e.g. GravNetHead Passes features per vertex through NN to compute new features & coordinates of vertex in latent space. Vertex then receives additional features based on aggregations of distance-weighted features for k-nearest vertices in latent space Second NN transforms features per vertex. Input (batch x vertices x features) –> Output (batch x vertices x new features)

Parameters

- **n_fpv** (int) – number of features per vertex to expect
- **n_s** (int) – number of latent-spatial dimensions to compute
- **n_lr** (int) – number of features to compute per vertex for latent representation
- **k** (int) – number of neighbours (including self) each vertex should consider when aggregating latent-representation features
- **agg_methods** (List[Callable[[Tensor], Tensor]]) – list of functions to use to aggregate distance-weighted latent-representation features
- **n_out** (int) – number of output features to compute per vertex

- **cat_means** (bool) – if True, will extend the incoming features per vertex by including the means of all features across all vertices GNNHead aslo has a *cat_means* argument, which should be set to *False* if enabled here (otherwise averaging happens twice).
- **f_slr_depth** (int) – number of layers to use for the latent rep. NN
- **f_out_depth** (int) – number of layers to use for the output NN
- **potential** (Callable[[Tensor], Tensor]) – function to control distance weighting (default is the exp(-d^2) potential used in the paper)
- **use_sa** (bool) – if true, will apply self-attention layer to the neighbourhhood features per vertex prior to aggregation
- **do** (float) – dropout rate to be applied to hidden layers in the NNs
- **bn** (bool) – whether batch normalisation should be applied to hidden layers in the NNs
- **act** (str) – activation function to apply to hidden layers in the NNs
- **lookup_init** (Callable[[str, Optional[int], Optional[int]], Callable[[Tensor], None]]) – function taking choice of activation function, number of inputs, and number of outputs an returning a function to initialise layer weights.
- **lookup_act** (Callable[[str], Any]) – function taking choice of activation function and returning an activation function layer
- **freeze** – whether to start with module parameters set to untrainable
- **bn_class** (Callable[[int], Module]) – class to use for BatchNorm, default is *LCBatchNorm1d*
- **sa_class** (Callable[[int], Module]) – class to use for self-attention layers, default is *SelfAttention*

forward(*x*)

Pass batch of vertices through GravNet layer and return new features per vertex

Parameters **x** (Tensor) – Incoming data (batch x vertices x features)

Return type Tensor

Returns Data with new features per vertex (batch x vertices x new features)

get_out_size()

Return type int

lumin.nn.models.blocks.head module

class lumin.nn.models.blocks.head.**CatEmbHead**(*cont_feats*, *do_cont=0*, *do_cat=0*, *cat_embedder=None*, *lookup_init=<function lookup_normal_init>*, *freeze=False*)

Bases: lumin.nn.models.blocks.head.AbsHead

Standard model head for columnar data. Provides inputs for continuous features and embedding matrices for categorical inputs, and uses a dense layer to upscale to width of network body. Designed to be passed as a 'head' to *ModelBuilder*. Supports batch normalisation and dropout (at separate rates for continuous features and categorical embeddings). Continuous features are expected to be the first len(cont_feats) columns of input

tensors and categorical features the remaining columns. Embedding arguments for categorical features are set using a *CatEmbedder*.

Parameters

- **cont_feats** (List[str]) – list of names of continuous input features
- **do_cont** (float) – if not None will add a dropout layer with dropout rate do acting on the continuous inputs prior to concatination wih the categorical embeddings
- **do_cat** (float) – if not None will add a dropout layer with dropout rate do acting on the categorical embeddings prior to concatination wih the continuous inputs
- **cat_embedder** (Optional[*CatEmbedder*]) – *CatEmbedder* providing details of how to embed categorical inputs
- **lookup_init** (Callable[[str, Optional[int], Optional[int]], Callable[[Tensor], None]]) – function taking choice of activation function, number of inputs, and number of outputs an returning a function to initialise layer weights.
- **freeze** (bool) – whether to start with module parameters set to untrainable

Examples::

```
>>> head = CatEmbHead(cont_feats=cont_feats)
>>>
>>> head = CatEmbHead(cont_feats=cont_feats,
...                   cat_embedder=CatEmbedder.from_fy(train_fy))
>>>
>>> head = CatEmbHead(cont_feats=cont_feats,
...                   cat_embedder=CatEmbedder.from_fy(train_fy),
...                   do_cont=0.1, do_cat=0.05)
>>>
>>> head = CatEmbHead(cont_feats=cont_feats,
...                   cat_embedder=CatEmbedder.from_fy(train_fy),
...                   lookup_init=lookup_uniform_init)
```

forward(*x*)

Pass tensor through block

Parameters **x** (Tensor) – input tensor

Returns Resulting tensor

Return type Tensor

get_embeds()

Get state_dict for every embedding matrix.

Return type Dict[str, OrderedDict]

Returns Dictionary mapping categorical features to learned embedding matrix

get_out_size()

Get size width of output layer

Return type int

Returns Width of output layer

plot_embeds(*savename=None*, *settings=<lumin.plotting.plot_settings.PlotSettings object>*)

Plot representations of embedding matrices for each categorical feature.

Parameters

- **savename** (Optional[str]) – if not None, will save copy of plot to give path
- **settings** (*PlotSettings*) – *PlotSettings* class to control figure appearance

Return type None

save_embeds(*path*)

Save learned embeddings to path. Each categorical embedding matic will be saved as a separate state_dict with name equal to the feature name as set in cat_embedder

Parameters **path** (Path) – path to which to save embedding weights

Return type None

class lumin.nn.models.blocks.head.**MultiHead**(*cont_feats*, *matrix_head*, *flat_head=<class 'lumin.nn.models.blocks.head.CatEmbHead'>*, *cat_embedder=None*, *lookup_init=<function lookup_normal_init>*, *freeze=False*, ***kargs*)

Bases: lumin.nn.models.blocks.head.AbsHead

Wrapper head to handel data containing flat continuous and categorical features, and matrix data. Flat inputs are passed through *flat_head*, and matrix inputs are passed through *matrix_head*. The outputs of both blocks are then concatenated together. Incoming data can either be: Completely flat, in which case the *matrix_head* should construct its own matrix from the data; or a tuple of flat data and the matrix, in which case the *matrix_head* will receive the data already in matrix format.

Parameters

- **cont_feats** (List[str]) – list of names of continuous and matrix input features
- **matrix_head** (Callable[[Any], AbsMatrixHead]) – Uninitialised (partial) head to handle matrix data e.g. InteractionNet
- **flat_head** (Callable[[Any], AbsHead]) – Uninitialised (partial) head to handle flat data e.g. *CatEmbHead*
- **cat_embedder** (Optional[*CatEmbedder*]) – *CatEmbedder* providing details of how to embed categorical inputs
- **lookup_init** (Callable[[str, Optional[int], Optional[int]], Callable[[Tensor], None]]) – function taking choice of activation function, number of inputs, and number of outputs an returning a function to initialise layer weights.
- **freeze** (bool) – whether to start with module parameters set to untrainable

Examples:: >>> inet = partial(InteractionNet, intfunc_depth=2,intfunc_width=4,intfunc_out_sz=3, ... outfunc_depth=2,outfunc_width=5,outfunc_out_sz=4,agg_method='flatten', ... feats_per_vec=feats_per_vec,vecs=vecs, act='swish') ... multihead = MultiHead(cont_feats=cont_feats+matrix_feats, matrix_head=inet, cat_embedder=CatEmbedder.from_fy(train_fy))

forward(*x*)

Pass incoming data through flat and matrix heads. If *x* is a *Tuple* then the first element is passed to the flat head and the secons is sent to the matrix head. Else the elements corresponding to flat dta are sent to the flat head and the elements corresponding to matrix elements are sent to the matrix head.

Parameters **x** (Union[Tensor, Tuple[Tensor, Tensor]]) – input data as either a flat *Tensor* or a *Tuple* of the form *[flat Tensor, matrix Tensor]*

Return type Tensor

Returns Concetanted outout of flat and matrix heads

get_out_size()
Get size of output

Return type int

Returns Output size of flat head + output size of matrix head

class lumin.nn.models.blocks.head.**GNNHead**(*cont_feats*, *vecs*, *feats_per_vec*, *extractor*, *collapser*, *use_in_bn=False*, *cat_means=False*, *freeze=False*, *bn_class=<class 'torch.nn.modules.batchnorm.BatchNorm1d'>*, ***kargs*)
Bases: lumin.nn.models.blocks.head.AbsMatrixHead

Encasulating class for applying graph neural-networks to features per vertex. New features are extracted per vertex via a AbsGraphFeatExtractor, and then data is flattened via *GraphCollapser*

Incoming data can either be flat, in which case it is reshaped into a matrix, or be supplied directly into matrix form. Reshaping (row-wise or column-wise) depends on the *row_wise* class attribute of the feature extractor. Data will be automatically converted to row-wise for processing by the grpah collaser.

Note: To allow for the fact that there may be nonexistant features (e.g. z-component of missing energy), *cont_feats* should be a list of all matrix features which really do exist (i.e. are present in input data), and be in the same order as the incoming data. Nonexistant features will be set zero.

Parameters

- **cont_feats** (List[str]) – list of all the matrix features which are present in the input data
- **vecs** (List[str]) – list of objects, i.e. feature prefixes
- **feats_per_vec** (List[str]) – list of features per vertex, i.e. feature suffixes
- **use_int_bn** – If true, will apply batch norm to incoming features
- **cat_means** (bool) – if True, will extend the incoming features per vertex by including the means of all features across all vertices
- **extractor** (Callable[[Any], AbsGraphFeatExtractor]) – The AbsGraphFeatExtractor class to instantiate to create new features per vertex
- **collasper** – The *GraphCollapser* class to instantiate to collapse graph to flat data (batch x features)
- **freeze** (bool) – whether to start with module parameters set to untrainable
- **bn_class** (Callable[[int], Module]) – class to use for BatchNorm, default is *nn.BatchNorm1d*

forward(*x*)
Passes input through the GravNet head and returns a flat tensor.

Parameters **x** (Union[Tensor, Tuple[Tensor, Tensor]]) – If a tuple, the second element is assumed to the be the matrix data. If a flat tensor, will convert the data to a matrix

Return type Tensor

Returns Resulting tensor

get_out_size()
Get size of output

Return type int

Returns Width of output representation

class lumin.nn.models.blocks.head.**RecurrentHead**(*cont_feats*, *vecs*, *feats_per_vec*, *depth*, *width*, *bidirectional=False*, *rnn=<class 'torch.nn.modules.rnn.RNN'>*, *do=0.0*, *act='tanh'*, *stateful=False*, *freeze=False*, ***kargs*)
Bases: lumin.nn.models.blocks.head.AbsMatrixHead

Recurrent head for row-wise matrix data applying e.g. RNN, LSTM, GRU.

Incoming data can either be flat, in which case it is reshaped into a matrix, or be supplied directly into matrix form. Matrices should/will be row-wise: each column is a seperate object (e.g. particle and jet) and each row is a feature (e.g. energy and mometum component). Matrix elements are expected to be named according to *{object}_{feature}*, e.g. *photon_energy*. *vecs* (vectors) should then be a list of objects, i.e. row headers, feature prefixes. *feats_per_vec* should be a list of features, i.e. column headers, feature suffixes.

Note: To allow for the fact that there may be nonexistant features (e.g. z-component of missing energy), *cont_feats* should be a list of all matrix features which really do exist (i.e. are present in input data), and be in the same order as the incoming data. Nonexistant features will be set zero.

Parameters

- **cont_feats** (List[str]) – list of all the matrix features which are present in the input data
- **vecs** (List[str]) – list of objects, i.e. row headers, feature prefixes
- **feats_per_vec** (List[str]) – list of features per object, i.e. columns headers, feature suffixes
- **depth** (int) – number of hidden layers to use
- **width** (int) – size of each hidden state
- **bidirectional** (bool) – whether to set recurrent layers to be bidirectional
- **rnn** (RNNBase) – module class to use for the recurrent layer, e.g. *torch.nn.RNN*, *torch.nn.LSTM*, *torch.nn.GRU*
- **do** (float) – dropout rate to be applied to hidden layers
- **act** (str) – activation function to apply to hidden layers, only used if rnn expects a nonliearity
- **stateful** (bool) – whether to return all intermediate hidden states, or only the final hidden states
- **freeze** (bool) – whether to start with module parameters set to untrainable

Examples::

```
>>> rnn = RecurrentHead(cont_feats=matrix_feats, feats_per_vec=feats_per_vec,
↪vecs=vecs, depth=1, width=20)
>>>
>>> rnn = RecurrentHead(cont_feats=matrix_feats, feats_per_vec=feats_per_vec,
↪vecs=vecs,
...                      depth=2, width=10, act='relu', bidirectional=True)
>>>
>>> lstm = RecurrentHead(cont_feats=matrix_feats, feats_per_vec=feats_per_
↪vec,vecs=vecs,
...                       depth=1, width=10, rnn=nn.LSTM)
>>>
>>> gru = RecurrentHead(cont_feats=matrix_feats, feats_per_vec=feats_per_vec,
↪vecs=vecs,
...                      depth=3, width=10, rnn=nn.GRU, bidirectional=True)
```

forward(*x*)

Passes input through the recurrent network.

Parameters **x** (Union[Tensor, Tuple[Tensor, Tensor]]) – If a tuple, the second element is assumed to the be the matrix data. If a flat tensor, will conver the data to a matrix

Return type Tensor

Returns if stateful, returns all hidden states, otherwise only returns last hidden state

get_out_size()

Get size of output

Return type Union[int, Tuple[int, int]]

Returns Width of output representation, or shape of output if stateful

class lumin.nn.models.blocks.head.**AbsConv1dHead**(*cont_feats*, *vecs*, *feats_per_vec*, *act='relu'*, *bn=False*, *layer_kargs=None*, *lookup_init=<function lookup_normal_init>*, *lookup_act=<function lookup_act>*, *freeze=False*, *bn_class=<class 'torch.nn.modules.batchnorm.BatchNorm1d'>*, ***kargs*)

Bases: lumin.nn.models.blocks.head.AbsMatrixHead

Abstract wrapper head for applying 1D convolutions to column-wise matrix data. Users should inherit from this class and overload get_layers() to define their model. Some common convolutional layers are already defined (e.g. ConvBlock and ResNeXt), which are accessable using methods such as :meth‘~lumin.nn.models.blocks.heads.AbsConv1dHead..get_conv1d_block‘. For more complicated models, foward() can also be overwritten The output size of the block is automatically computed during initialisation by passing through random pseudodata.

Incoming data can either be flat, in which case it is reshaped into a matrix, or be supplied directly into matrix form. Matrices should/will be row-wise: each column is a seperate object (e.g. particle and jet) and each row is a feature (e.g. energy and mometum component). Matrix elements are expected to be named according to *{object}_{feature}*, e.g. *photon_energy*. *vecs* (vectors) should then be a list of objects, i.e. row headers, feature prefixes. *feats_per_vec* should be a list of features, i.e. column headers, feature suffixes.

Note: To allow for the fact that there may be nonexistant features (e.g. z-component of missing energy),

cont_feats should be a list of all matrix features which really do exist (i.e. are present in input data), and be in the same order as the incoming data. Nonexistant features will be set zero.

Parameters

- **cont_feats** (List[str]) – list of all the matrix features which are present in the input data
- **vecs** (List[str]) – list of objects, i.e. row headers, feature prefixes
- **feats_per_vec** (List[str]) – list of features per object, i.e. columns headers, feature suffixes
- **act** (str) – activation function passed to *get_layers*
- **bn** (bool) – batch normalisation argument passed to *get_layers*
- **layer_kargs** (Optional[Dict[str, Any]]) – dictionary of keyword arguments which are passed to *get_layers*
- **lookup_init** (Callable[[str, Optional[int], Optional[int]], Callable[[Tensor], None]]) – function taking choice of activation function, number of inputs, and number of outputs an returning a function to initialise layer weights.
- **freeze** (bool) – whether to start with module parameters set to untrainable
- **bn_class** (Callable[[int], Module]) – class to use for BatchNorm, default is *nn.BatchNorm1d*

Examples::

```
>>> class MyCNN(AbsConv1dHead):
...     def get_layers(self, act:str='relu', bn:bool=False, **kargs) ->␣
↪Tuple[nn.Module, int]:
...         layers = []
...         layers.append(self.get_conv1d_block(3, 16, stride=1, kernel_sz=3,
↪ act=act, bn=bn))
...         layers.append(self.get_conv1d_block(16, 16, stride=1, kernel_
↪sz=3, act=act, bn=bn))
...         layers.append(self.get_conv1d_block(16, 32, stride=2, kernel_
↪sz=3, act=act, bn=bn))
...         layers.append(self.get_conv1d_block(32, 32, stride=1, kernel_
↪sz=3, act=act, bn=bn))
...         layers.append(nn.AdaptiveAvgPool1d(1))
...         layers = nn.Sequential(*layers)
...         return layers
...
... cnn = MyCNN(cont_feats=matrix_feats, vecs=vectors, feats_per_vec=feats_
↪per_vec)
>>>
>>> class MyResNet(AbsConv1dHead):
...     def get_layers(self, act:str='relu', bn:bool=False, **kargs) ->␣
↪Tuple[nn.Module, int]:
...         layers = []
...         layers.append(self.get_conv1d_block(3, 16, stride=1, kernel_sz=3,
↪ act='linear', bn=False))
...         layers.append(self.get_conv1d_res_block(16, 16, stride=1, kernel_
↪sz=3, act=act, bn=bn))
...         layers.append(self.get_conv1d_res_block(16, 32, stride=2, kernel_
↪sz=3, act=act, bn=bn))
```

(continues on next page)

(continued from previous page)

```
...         layers.append(self.get_conv1d_res_block(32, 32, stride=1, kernel_
↪sz=3, act=act, bn=bn))
...         layers.append(nn.AdaptiveAvgPool1d(1))
...         layers = nn.Sequential(*layers)
...         return layers
...
... cnn = MyResNet(cont_feats=matrix_feats, vecs=vectors, feats_per_
↪vec=feats_per_vec)
>>>
>>> class MyResNeXt(AbsConv1dHead):
...     def get_layers(self, act:str='relu', bn:bool=False, **kargs) ->␣
↪Tuple[nn.Module, int]:
...         layers = []
...         layers.append(self.get_conv1d_block(3, 32, stride=1, kernel_sz=3,
↪ act='linear', bn=False))
...         layers.append(self.get_conv1d_resNeXt_block(32, 4, 4, 32,␣
↪stride=1, kernel_sz=3, act=act, bn=bn))
...         layers.append(self.get_conv1d_resNeXt_block(32, 4, 4, 32,␣
↪stride=2, kernel_sz=3, act=act, bn=bn))
...         layers.append(self.get_conv1d_resNeXt_block(32, 4, 4, 32,␣
↪stride=1, kernel_sz=3, act=act, bn=bn))
...         layers.append(nn.AdaptiveAvgPool1d(1))
...         layers = nn.Sequential(*layers)
...         return layers
...
... cnn = MyResNeXt(cont_feats=matrix_feats, vecs=vectors, feats_per_
↪vec=feats_per_vec)
```

check_out_sz()

Automatically computes the output size of the head by passing through random data of the expected shape

Return type int

Returns x.size(-1) where x is the outgoing tensor from the head

forward(*x*)

Passes input through the convolutional network.

Parameters **x** (Union[Tensor, Tuple[Tensor, Tensor]]) – If a tuple, the second element is assumed to the be the matrix data. If a flat tensor, will conver the data to a matrix

Return type Tensor

Returns Resulting tensor

get_conv1d_block(*in_c*, *out_c*, *kernel_sz*, *padding='auto'*, *stride=1*, *act='relu'*, *bn=False*)

Wrapper method to build a ConvBlock object.

Parameters

- **in_c** (int) – number of input channels (number of features per object / rows in input matrix)
- **out_c** (int) – number of output channels (number of features / rows in output matrix)
- **kernel_sz** (int) – width of kernel, i.e. the number of columns to overlay
- **padding** (Union[int, str]) – amount of padding columns to add at start and end of convolution. If left as 'auto', padding will be automatically computed to

conserve the number of columns.

- **stride** (int) – number of columns to move kernel when computing convolutions. Stride 1 = kernel centred on each column, stride 2 = kernel centred on ever other column and input size halved, et cetera.
- **act** (str) – string representation of argument to pass to lookup_act
- **bn** (bool) – whether to use batch normalisation (order is weights->activation->batchnorm)

Return type *Conv1DBlock*

Returns Instantiated ConvBlock object

get_conv1d_resNeXt_block(*in_c*, *inter_c*, *cardinality*, *out_c*, *kernel_sz*, *padding='auto'*, *stride=1*, *act='relu'*, *bn=False*)

Wrapper method to build a *ResNeXt1DBlock* object.

Parameters

- **in_c** (int) – number of input channels (number of features per object / rows in input matrix)
- **inter_c** (int) – number of intermediate channels in groups
- **cardinality** (int) – number of groups
- **out_c** (int) – number of output channels (number of features / rows in output matrix)
- **kernel_sz** (int) – width of kernel, i.e. the number of columns to overlay
- **padding** (Union[int, str]) – amount of padding columns to add at start and end of convolution. If left as 'auto', padding will be automatically computed to conserve the number of columns.
- **stride** (int) – number of columns to move kernel when computing convolutions. Stride 1 = kernel centred on each column, stride 2 = kernel centred on ever other column and input size halved, et cetera.
- **act** (str) – string representation of argument to pass to lookup_act
- **bn** (bool) – whether to use batch normalisation (order is pre-activation: batchnorm->activation->weights)

Return type *ResNeXt1DBlock*

Returns Instantiated *ResNeXt1DBlock* object

get_conv1d_res_block(*in_c*, *out_c*, *kernel_sz*, *padding='auto'*, *stride=1*, *act='relu'*, *bn=False*)

Wrapper method to build a *Res1DBlock* object.

Parameters

- **in_c** (int) – number of input channels (number of features per object / rows in input matrix)
- **out_c** (int) – number of output channels (number of features / rows in output matrix)
- **kernel_sz** (int) – width of kernel, i.e. the number of columns to overlay
- **padding** (Union[int, str]) – amount of padding columns to add at start and end of convolution. If left as 'auto', padding will be automatically computed to conserve the number of columns.

- **stride** (int) – number of columns to move kernel when computing convolutions. Stride 1 = kernel centred on each column, stride 2 = kernel centred on ever other column and input size halved, et cetera.
- **act** (str) – string representation of argument to pass to lookup_act
- **bn** (bool) – whether to use batch normalisation (order is pre-activation: batchnorm->activation->weights)

Return type *Res1DBlock*

Returns Instantiated *Res1DBlock* object

abstract get_layers(*in_c*, *act='relu'*, *bn=False*, ***kargs*)

Abstract function to be overloaded by user. Should return a single torch.nn.Module which accepts the expected input matrix data.

Return type Module

get_out_size()

Get size of output

Return type int

Returns Width of output representation

class lumin.nn.models.blocks.head.**LorentzBoostNet**(*cont_feats*, *vecs*, *feats_per_vec*, *n_particles*, *feat_extractor=None*, *bn=True*, *lookup_init=<function lookup_normal_init>*, *lookup_act=<function lookup_act>*, *freeze=False*, *bn_class=<class 'torch.nn.modules.batchnorm.BatchNorm1d'>*, ***kargs*)

Bases: lumin.nn.models.blocks.head.AbsMatrixHead

Implementation of the Lorentz Boost Network (https://arxiv.org/abs/1812.09722), which takes 4-momenta for particles and learns new particles and reference frames from linear combinations of the original particles, and then boosts the new particles into the learned reference frames. Preset kernel functions are the run over the 4-momenta of the boosted particles to compute a set of veriables per particle. These functions can be based on pairs etc. of particles, e.g. angles between particles. (*LorentzBoostNet.comb* provides an index iterator over all paris of particles).

A default feature extractor is provided which returns the (px,py,pz,E) of the boosted particles and the cosine angle between every pair of boosted particle. This can be overwritten by passing a function to the *feat_extractor* argument during initialisation, or overidding *LorentzBoostNet.feat_extractor*.

Important: 4-momenta should be supplied without preprocessing, and 4-momenta must be physical (E>=|p|). It is up to the user to ensure this, and not doing so may result in errors. A BatchNorm argument (*bn*) is available to preprocess the features extracted from the boosted particles prior to returning them.

Incoming data can either be flat, in which case it is reshaped into a matrix, or be supplied directly in row-wise matrix form. Matrices should/will be row-wise: each row is a seperate 4-momenta in the form (px,py,pz,E). Matrix elements are expected to be named according to *{particle}_{feature}*, e.g. *photon_E*. *vecs* (vectors) should then be a list of particles, i.e. row headers, feature prefixes. *feats_per_vec* should be a list of features, i.e. column headers, feature suffixes.

Note: To allow for the fact that there may be nonexistant features (e.g. z-component of missing energy), *cont_feats* should be a list of all matrix features which really do exist (i.e. are present in input data), and be in the same order as the incoming data. Nonexistant features will be set zero.

Parameters

- **cont_feats** (List[str]) – list of all the matrix features which are present in the input data
- **vecs** (List[str]) – list of objects, i.e. row headers, feature prefixes
- **feats_per_vec** (List[str]) – list of features per object, i.e. column headers, feature suffixes
- **n_particles** (int) – the number of particles and reference frames to learn
- **feat_extractor** (Optional[Callable[[Tensor], Tensor]]) – if not None, will use the argument as the function to extract features from the 4-momenta of the boosted particles.
- **bn** (bool) – whether batch normalisation should be applied to the extracted features
- **lookup_init** (Callable[[str, Optional[int], Optional[int]], Callable[[Tensor], None]]) – function taking choice of activation function, number of inputs, and number of outputs an returning a function to initialise layer weights. Purely for inheritance, unused by class as is.
- **lookup_act** (Callable[[str], Any]) – function taking choice of activation function and returning an activation function layer. Purely for inheritance, unused by class as is.
- **freeze** (bool) – whether to start with module parameters set to untrainable.
- **bn_class** (Callable[[int], Module]) – class to use for BatchNorm, default is *nn.BatchNorm1d*

Examples::

```
>>> lbn = LorentzBoostNet(cont_feats=matrix_feats, feats_per_vec=feats_per_
↪vec,vecs=vecs, n_particles=6)
>>>
>>> def feat_extractor(x:Tensor) -> Tensor:  # Return masses of boosted␣
↪particles, x dimensions = [batch,particle,4-mom]
...     momenta,energies =  x[:,:,:3], x[:,:,3:]
...     mass = torch.sqrt((energies**2)-torch.sum(momenta**2, dim=-1)[:,:,
↪None])
...     return mass
>>> lbn = InteractionNet(cont_feats=matrix_feats, feats_per_vec=feats_per_
↪vec,vecs=vecs, n_particle=6, feat_extractor=feat_extractor)
```

check_out_sz()

Automatically computes the output size of the head by passing through random data of the expected shape

Return type int

Returns x.size(-1) where x is the outgoing tensor from the head

feat_extractor(*x*)

Computes features from boosted particle 4-momenta. Incoming tensor *x* contains all 4-momenta for all particles for all datapoints in minibatch. Default function returns 4-momenta and cosine angle between all particles.

Parameters **x** (Tensor) – 3D incoming tensor with dimensions: [batch, particle, 4-mom (px,py,pz,E)]

Return type Tensor

Returns 2D tensor with dimensions [batch, features]

forward(*x*)

Passes input through the LB network and aggregates down to a flat tensor via the feature extractor, optionally passing through a batchnorm layer.

Parameters **x** (Union[Tensor, Tuple[Tensor, Tensor]]) – If a tuple, the second element is assumed to the be the matrix data. If a flat tensor, will convert the data to a matrix

Return type Tensor

Returns Resulting tensor

get_out_size()

Get size of output

Return type int

Returns Width of output representation

class lumin.nn.models.blocks.head.**AutoExtractLorentzBoostNet**(*cont_feats*, *vecs*, *feats_per_vec*, *n_particles*, *depth*, *width*, *n_singles=0*, *n_pairs=0*, *act='swish'*, *do=0*, *bn=False*, *lookup_init=<function lookup_normal_init>*, *lookup_act=<function lookup_act>*, *freeze=False*, *bn_class=<class 'torch.nn.modules.batchnorm.BatchNorm1d'>*, ***kargs*)

Bases: *lumin.nn.models.blocks.head.LorentzBoostNet*

Modified version of :class:‘~lumin.nn.models.blocks.head.LorentzBoostNet (implementation of the Lorentz Boost Network (https://arxiv.org/abs/1812.09722)). Rather than relying on fixed kernel functions to extract features from the boosted paricles, the functions are learnt during training via neural networks.

Two netrowks are used, one to extract *n_singles* features from each particle and another to extract *n_pairs* features from each pair of particles.

Important: 4-momenta should be supplied without preprocessing, and 4-momenta must be physical (E>=|p|). It is up to the user to ensure this, and not doing so may result in errors. A BatchNorm argument (*bn*) is available to preprocess the 4-momenta of the boosted particles prior to passing them through the neural networks

---

Incoming data can either be flat, in which case it is reshaped into a matrix, or be supplied directly in row-wise matrix form. Matrices should/will be row-wise: each row is a seperate 4-momenta in the form (px,py,pz,E). Matrix elements are expected to be named according to *{particle}_{feature}*, e.g. *photon_E*. *vecs* (vectors)

should then be a list of particles, i.e. row headers, feature prefixes. *feats_per_vec* should be a list of features, i.e. column headers, feature suffixes.

Note: To allow for the fact that there may be nonexistant features (e.g. z-component of missing energy), *cont_feats* should be a list of all matrix features which really do exist (i.e. are present in input data), and be in the same order as the incoming data. Nonexistant features will be set zero.

Parameters

- **cont_feats** (List[str]) – list of all the matrix features which are present in the input data
- **vecs** (List[str]) – list of objects, i.e. column headers, feature prefixes
- **feats_per_vec** (List[str]) – list of features per object, i.e. row headers, feature suffixes
- **n_particles** (int) – the number of particles and reference frames to learn
- **depth** (int) – the number of hidden layers in each network
- **width** (int) – the number of neurons per hidden layer
- **n_singles** (int) – the number of features to extract per individual particle
- **n_pairs** (int) – the number of features to extract per pair of particles
- **act** (str) – string representation of argument to pass to lookup_act. Activation should ideally have non-zero outputs to help deal with poorly normalised inputs
- **do** (float) – dropout rate for use in networks
- **bn** (bool) – whether to use batch normalisation within networks. Inputs are passed through BN regardless of setting.
- **lookup_init** (Callable[[str, Optional[int], Optional[int]], Callable[[Tensor], None]]) – function taking choice of activation function, number of inputs, and number of outputs an returning a function to initialise layer weights.
- **lookup_act** (Callable[[str], Any]) – function taking choice of activation function and returning an activation function layer.
- **freeze** (bool) – whether to start with module parameters set to untrainable.
- **bn_class** (Callable[[int], Module]) – class to use for BatchNorm, default is *nn.BatchNorm1d*

Examples::

```
>>> aelbn = AutoExtractLorentzBoostNet(cont_feats=matrix_feats, feats_per_
↪vec=feats_per_vec,vecs=vecs, n_particles=6,
                                       depth=3, width=10, n_singles=3, n_
↪pairs=2)
```

feat_extractor(*x*)

Computes features from boosted particle 4-momenta. Incoming tensor *x* contains all 4-momenta for all particles for all datapoints in minibatch. *single_nn* broadcast to all boosted particles, and *pair_nn* broadcast to all paris of particles. Returned features are concatenated together.

Parameters **x** (Tensor) – 3D incoming tensor with dimensions: [batch, particle, 4-mom (px,py,pz,E)]

Return type Tensor

Returns 2D tensor with dimensions [batch, features]

lumin.nn.models.blocks.tail module

class lumin.nn.models.blocks.tail.**IdentTail**(*n_in*, *n_out*, *objective*, *bias_init=None*, *lookup_init=<function lookup_normal_init>*, *freeze=False*)

Bases: lumin.nn.models.blocks.tail.AbsTail

Placeholder tail module for cases in which a tail is not required. Outputs are equal to imputs.

forward(*x*)

Pass tensor through block

Parameters **x** (Tensor) – input tensor

Returns Resulting tensor

Return type Tensor

get_out_size()

Get size width of output layer

Return type int

Returns Width of output layer

class lumin.nn.models.blocks.tail.**ClassRegMulti**(*n_in*, *n_out*, *objective*, *y_range=None*, *bias_init=None*, *y_mean=None*, *y_std=None*, *lookup_init=<function lookup_normal_init>*, *freeze=False*)

Bases: lumin.nn.models.blocks.tail.AbsTail

Output block for (multi(class/label)) classification or regression tasks. Designed to be passed as a 'tail' to *ModelBuilder*. Takes output size of network body and scales it to required number of outputs. For regression tasks, y_range can be set with per-output minima and maxima. The outputs are then adjusted according to ((y_max-y_min)*x)+self.y_min, where x is the output of the network passed through a sigmoid function. Effectively allowing regression to be performed without normalising and standardising the target values. Note it is safest to allow some leaway in setting the min and max, e.g. max = 1.2*max, min = 0.8*min Output activation function is automatically set according to objective and y_range.

Parameters

- **n_in** (int) – number of inputs to expect
- **n_out** (int) – number of outputs required
- **objective** (str) – string representation of network objective, i.e. 'classification', 'regression', 'multiclass'
- **y_range** (Union[Tuple, ndarray, None]) – if not None, will apply rescaling to network outputs: x = ((y_range[1]-y_range[0])*sigmoid(x))+y_range[0]. Incompatible with *y_mean* and *y_std*
- **bias_init** (Optional[float]) – specify an intial bias for the output neurons. Otherwise default values of 0 are used, except for multiclass objectives, which use 1/n_out

- **y_mean** (Union[float, List[float], ndarray, None]) – if sepcified along with *y_std*, will apply rescaling to network outputs: x = (y_std*x)+y_mean. Incopmpatible with *y_range*
- **y_std** (Union[float, List[float], ndarray, None]) – if sepcified along with *y_mean*, will apply rescaling to network outputs: x = (y_std*x)+y_mean. Incopmpatible with *y_range*
- **lookup_init** (Callable[[str, Optional[int], Optional[int]], Callable[[Tensor], None]]) – function taking string representation of activation function, number of inputs, and number of outputs an returning a function to initialise layer weights.

Examples::

```
>>> tail = ClassRegMulti(n_in=100, n_out=1, objective='classification')
>>>
>>> tail = ClassRegMulti(n_in=100, n_out=5, objective='multiclass')
>>>
>>> y_range = (0.8*targets.min(), 1.2*targets.max())
>>> tail = ClassRegMulti(n_in=100, n_out=1, objective='regression',
...                      y_range=y_range)
>>>
>>> min_targs = np.min(targets, axis=0).reshape(targets.shape[1],1)
>>> max_targs = np.max(targets, axis=0).reshape(targets.shape[1],1)
>>> min_targs[min_targs > 0] *=0.8
>>> min_targs[min_targs < 0] *=1.2
>>> max_targs[max_targs > 0] *=1.2
>>> max_targs[max_targs < 0] *=0.8
>>> y_range = np.hstack((min_targs, max_targs))
>>> tail = ClassRegMulti(n_in=100, n_out=6, objective='regression',
...                      y_range=y_range,
...                      lookup_init=lookup_uniform_init)
```

forward(*x*)

Pass tensor through block

Parameters **x** (Tensor) – input tensor

Returns Resulting tensor

Return type Tensor

get_out_size()

Get size width of output layer

Return type int

Returns Width of output layer

Module contents

lumin.nn.models.layers package

Submodules

lumin.nn.models.layers.activations module

lumin.nn.models.layers.activations.**lookup_act**(*act*)

Map activation name to class

Parameters **act** (str) – string representation of activation function

Return type Any

Returns Class implementing requested activation function

class lumin.nn.models.layers.activations.**Swish**(*inplace=False*)

Bases: torch.nn.modules.module.Module

Non-trainable Swish activation function https://arxiv.org/abs/1710.05941

Parameters **inplace** – whether to apply activation inplace

Examples::

```
>>> swish = Swish()
```

forward(*x*)

Pass tensor through Swish function

Parameters **x** (Tensor) – incoming tensor

Return type Tensor

Returns Resulting tensor

lumin.nn.models.layers.batchnorms module

class lumin.nn.models.layers.batchnorms.**LCBatchNorm1d**(*bn*)

Bases: torch.nn.modules.module.Module

Wrapper class for 1D batchnorm to make it run over (Batch x length x channel) data for use in NNs designed to be broadcast across matrix data.

Parameters **bn** (BatchNorm1d) – base 1D batchnorm module to call

forward(*x*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

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

Return type Tensor

class lumin.nn.models.layers.batchnorms.**RunningBatchNorm1d**(*nf*, *mom=0.1*, *n_warmup=20*, *eps=1e-05*)

Bases: torch.nn.modules.module.Module

1D Running batchnorm implementation from fastai (https://github.com/fastai/course-v3) distributed under apache2 licence. Modifcations: Adaptation to 1D & 3D, add eps in mom1 calculation, type hinting, docs

Parameters

- **nf** (int) – number of features/channels

- **mom** (float) – momentum (fraction to add to running averages)
- **n_warmup** (int) – number of warmup iterations (during which variance is clamped)
- **eps** (float) – epsilon to prevent division by zero

forward(*x*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

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

Return type Tensor

update_stats(*x*)

Return type None

class lumin.nn.models.layers.batchnorms.**RunningBatchNorm2d**(*nf*, *mom=0.1*, *n_warmup=20*, *eps=1e-05*)

Bases: *lumin.nn.models.layers.batchnorms.RunningBatchNorm1d*

2D Running batchnorm implementation from fastai (https://github.com/fastai/course-v3) distributed under apache2 licence. Modifcations: add eps in mom1 calculation, type hinting, docs

Parameters

- **nf** (int) – number of features/channels
- **mom** (float) – momentum (fraction to add to running averages)
- **eps** (float) – epsilon to prevent division by zero

forward(*x*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

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

Return type Tensor

class lumin.nn.models.layers.batchnorms.**RunningBatchNorm3d**(*nf*, *mom=0.1*, *n_warmup=20*, *eps=1e-05*)

Bases: *lumin.nn.models.layers.batchnorms.RunningBatchNorm2d*

3D Running batchnorm implementation from fastai (https://github.com/fastai/course-v3) distributed under apache2 licence. Modifcations: Adaptation to 3D, add eps in mom1 calculation, type hinting, docs

Parameters

- **nf** (int) – number of features/channels
- **mom** (float) – momentum (fraction to add to running averages)

- **eps** (float) – epsilon to prevent division by zero

lumin.nn.models.layers.mish module

This file contains code modfied from https://github.com/digantamisra98/Mish which is made available under the following MIT Licence: MIT License

Copyright (c) 2019 Diganta Misra

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

The Apache Licence 2.0 underwhich the majority of the rest of LUMIN is distributed does not apply to the code within this file.

class lumin.nn.models.layers.mish.**Mish**

Bases: torch.nn.modules.module.Module

Applies the mish function element-wise: mish(x) = x * tanh(softplus(x)) = x * tanh(ln(1 + exp(x))) Shape:

- Input: (N, *) where * means, any number of additional dimensions
- Output: (N, *), same shape as the input

Examples

```
>>> m = Mish()
>>> input = torch.randn(2)
>>> output = m(input)
```

forward(*input*)

Forward pass of the function.

lumin.nn.models.layers.self_attention module

class lumin.nn.models.layers.self_attention.**SelfAttention**(*n_fpv*, *n_a*, *do=0*, *bn=False*, *act='relu'*, *lookup_init=<function lookup_normal_init>*, *lookup_act=<function lookup_act>*, *bn_class=<class 'torch.nn.modules.batchnorm.BatchNorm1d'>*)

Bases: torch.nn.modules.module.Module

Class for applying self attention (Vaswani et al. 2017 (https://arxiv.org/abs/1706.03762)) to features per vertex.

Parameters

- **n_fpv** (int) – number of features per vertex to expect
- **n_a** (int) – width of self attention representation (paper recommends n_fpv//4)
- **do** (float) – dropout rate to be applied to hidden layers in the NNs
- **bn** (bool) – whether batch normalisation should be applied to hidden layers in the NNs
- **act** (str) – activation function to apply to hidden layers in the NNs
- **lookup_init** (Callable[[str, Optional[int], Optional[int]], Callable[[Tensor], None]]) – function taking choice of activation function, number of inputs, and number of outputs an returning a function to initialise layer weights.
- **lookup_act** (Callable[[str], Any]) – function taking choice of activation function and returning an activation function layer
- **bn_class** (Callable[[int], Module]) – class to use for BatchNorm, default is *LCBatchNorm1d*

forward(*x*)

Augments features per vertex

Arguemnts: x: incoming data (batch x vertices x features)

Return type Tensor

Returns augmented features (batch x vertices x new features)

get_out_size()

Return type int

Module contents

Submodules

lumin.nn.models.helpers module

class lumin.nn.models.helpers.**CatEmbedder**(*cat_names*, *cat_szs*, *emb_szs=None*, *max_emb_sz=50*, *emb_load_path=None*)

Bases: object

Helper class for embedding categorical features. Designed to be passed to *ModelBuilder*. Note that the classmethod *from_fy()* may be used to instantiate an *CatEmbedder* from a *FoldYielder*.

Parameters

- **cat_names** (List[str]) – list of names of catgorical features in order in which they will be passed as inputs columns
- **cat_szs** (List[int]) – list of cardinalities (number of unique elements) for each feature
- **emb_szs** (Optional[List[int]]) – Optional list of embedding sizes for each feature. If None, will use min(max_emb_sz, (1+sz)//2)
- **max_emb_sz** (int) – Maximum size of embedding if emb_szs is None

- **emb_load_path** (Union[Path, str, None]) – if not None, will cause *ModelBuilder* to attempt to load pretrained embeddings from path

Examples::

```
>>> cat_embedder = CatEmbedder(cat_names=['n_jets', 'channel'],
                               cat_szs=[5, 3])
>>>
>>> cat_embedder = CatEmbedder(cat_names=['n_jets', 'channel'],
                               cat_szs=[5, 3], emb_szs=[2, 2])
>>>
>>> cat_embedder = CatEmbedder(cat_names=['n_jets', 'channel'],
                               cat_szs=[5, 3], emb_szs=[2, 2],
                               emb_load_path=Path('weights'))
```

calc_emb_szs()
Method used to set sizes of embeddings for each categorical feature when no embedding sizes are explicitly passed Uses rule of thumb of min(50, (1+cardinality)/2)

Return type None

classmethod from_fy(*fy*, *emb_szs=None*, *max_emb_sz=50*, *emb_load_path=None*)
Instantiate an *CatEmbedder* from a *FoldYielder*, i.e. avoid having to pass cat_names and cat_szs.

Parameters

- **fy** (*FoldYielder*) – *FoldYielder* with training data
- **emb_szs** (Optional[List[int]]) – Optional list of embedding sizes for each feature. If None, will use min(max_emb_sz, (1+sz)//2)
- **max_emb_sz** (int) – Maximum size of embedding if emb_szs is None
- **emb_load_path** (Union[Path, str, None]) – if not None, will cause *ModelBuilder* to attempt to load pretrained embeddings from path

Returns *CatEmbedder*

Examples::

```
>>> cat_embedder = CatEmbedder.from_fy(train_fy)
>>>
>>> cat_embedder = CatEmbedder.from_fy(train_fy, emb_szs=[2, 2])
>>>
>>> cat_embedder = CatEmbedder.from_fy(
        train_fy, emb_szs=[2, 2],
        emb_load_path=Path('weights'))
```

lumin.nn.models.initialisations module

lumin.nn.models.initialisations.**lookup_normal_init**(*act*, *fan_in=None*, *fan_out=None*)
Lookup for weight initialisation using Normal distributions

Parameters

- **act** (str) – string representation of activation function
- **fan_in** (Optional[int]) – number of inputs to neuron
- **fan_out** (Optional[int]) – number of outputs from neuron

Return type Callable[[Tensor], None]

Returns Callable to initialise weight tensor

lumin.nn.models.initialisations.**lookup_uniform_init**(*act*, *fan_in=None*, *fan_out=None*)

Lookup weight initialisation using Uniform distributions

Parameters

- **act** (str) – string representation of activation function
- **fan_in** (Optional[int]) – number of inputs to neuron
- **fan_out** (Optional[int]) – number of outputs from neuron

Return type Callable[[Tensor], None]

Returns Callable to initialise weight tensor

lumin.nn.models.model module

class lumin.nn.models.model.**Model**(*model_builder=None*)

Bases: lumin.nn.models.abs_model.AbsModel

Wrapper class to handle training and inference of NNs created via a *ModelBuilder*. Note that saved models can be instantiated direcly via *from_save()* classmethod.

TODO: Improve mask description & user-friendlyness, change to indicate that 'masked' inputs are actually the ones which are used

Parameters **model_builder** (Optional[*ModelBuilder*]) – *ModelBuilder* which will construct the network, loss, optimiser, and input mask

Examples::

```
>>> model = Model(model_builder)
```

evaluate(*inputs*, *targets=None*, *weights=None*, *bs=None*)

Compute loss on provided data.

Parameters

- **inputs** (Union[ndarray, Tensor, Tuple, *BatchYielder*]) – input data, or *BatchYielder* with input, target, and weight data
- **targets** (Union[ndarray, Tensor, None]) – targets, not required if *BatchYielder* is passed to inputs
- **weights** (Union[ndarray, Tensor, None]) – Optional weights, not required if *BatchYielder*, or no weights should be considered
- **bs** (Optional[int]) – batch size to use. If *None*, will evaluate all data at once

Return type float

Returns (weighted) loss of model predictions on provided data

export2onnx(*name*, *bs=1*)

Export network to ONNX format. Note that ONNX expects a fixed batch size (bs) which is the number of datapoints your wish to pass through the model concurrently.

Parameters

- **name** (str) – filename for exported file

- **bs** (int) – batch size for exported models

Return type None

export2tfpb(*name*, *bs=1*)
Export network to Tensorflow ProtocolBuffer format, via ONNX. Note that ONNX expects a fixed batch size (bs) which is the number of datapoints your wish to pass through the model concurrently.

Parameters

- **name** (str) – filename for exported file
- **bs** (int) – batch size for exported models

Return type None

fit(*n_epochs*, *fy*, *bs*, *bulk_move=True*, *train_on_weights=True*, *trn_idxs=None*, *val_idx=None*, *cbs=None*, *cb_savepath=Path('train_weights')*, *model_bar=None*, *visible_bar=True*)
Fit network to training data according to the model's loss and optimiser.

Training continues until: - All of the training folds are used n_epoch number of times; - Or a callback triggers training to stop, e.g. *OneCycle*,

or *EarlyStopping*

Parameters

- **n_epochs** (int) – number of epochs for which to train
- **fy** (*FoldYielder*) – *FoldYielder* containing training and validation data
- **bs** (int) – Batch size
- **bulk_move** (bool) – if true, will optimise for speed by using more RAM and VRAM
- **train_on_weights** (bool) – whether to actually use data weights, if present
- **trn_idxs** (Optional[List[int]]) – Fold indexes in *fy* to use for training. If not set, will use all folds except val_idx
- **val_idx** (Optional[int]) – Fold index in *fy* to use for validation. If not set, will not compute validation losses
- **cbs** (Union[AbsCallback, List[AbsCallback], None]) – list of instantiated callbacks to adjust training. Will be called in order listed.
- **cb_savepath** (Path) – General save directory for any callbacks which require saving models and other information (accessible from *fit_params*),
- **model_bar** (Optional[ConsoleMasterBar]) – Optional *master_bar* for aligning progress bars, i.e. if training multiple models

Return type List[AbsCallback]

Returns List of all callbacks used during training

freeze_layers()
Make parameters untrainable

Return type None

classmethod from_save(*name*, *model_builder*)
Instantiated a *Model* and load saved state from file.

Parameters

- **name** (str) – name of file containing saved state
- **model_builder** (*ModelBuilder*) – *ModelBuilder* which was used to construct the network

Return type AbsModel

Returns Instantiated *Model* with network weights, optimiser state, and input mask loaded from saved state

Examples::

```
>>> model = Model.from_save('weights/model.h5', model_builder)
```

get_feat_importance(*fy*, *bs=None*, *eval_metric=None*, *savename=None*, *settings=<lumin.plotting.plot_settings.PlotSettings object>*)

Call *get_nn_feat_importance()* passing this *Model* and provided arguments

Parameters

- **fy** (*FoldYielder*) – *FoldYielder* interfacing to data used to train model
- **bs** (Optional[int]) – If set, will evaluate model in batches of data, rather than all at once
- **eval_metric** (Optional[*EvalMetric*]) – Optional EvalMetric to use to quantify performance in place of loss
- **savename** (Optional[str]) – Optional name of file to which to save the plot of feature importances
- **settings** (*PlotSettings*) – *PlotSettings* class to control figure appearance

Return type DataFrame

get_lr()

Get learning rate of optimiser

Return type float

Returns learning rate of optimiser

get_mom()

Get momentum/beta_1 of optimiser

Return type float

Returns momentum/beta_1 of optimiser

get_out_size()

Get number of outputs of model

Return type int

Returns Number of outputs of model

get_param_count(*trainable=True*)

Return number of parameters in model.

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

Return type int

Returns NUmber of (trainable) parameters in model

get_weights()
Get state_dict of weights for network

Return type OrderedDict

Returns state_dict of weights for network

load(*name*, *model_builder=None*)
Load model, optimiser, and input mask states from file

Parameters

- **name** (str) – name of save file
- **model_builder** (Optional[*ModelBuilder*]) – if *Model* was not initialised with a *ModelBuilder*, you will need to pass one here

Return type None

predict(*inputs*, *as_np=True*, *pred_name='pred'*, *pred_cb=<lumin.nn.callbacks.pred_handlers.PredHandler object>*, *cbs=None*, *bs=None*)
Apply model to inputed data and compute predictions.

Parameters

- **inputs** (Union[ndarray, DataFrame, Tensor, *FoldYielder*]) – input data as Numpy array, Pandas DataFrame, or tensor on device, or *FoldYielder* interfacing to data
- **as_np** (bool) – whether to return predictions as Numpy array (otherwise tensor) if inputs are a Numpy array, Pandas DataFrame, or tensor
- **pred_name** (str) – name of group to which to save predictions if inputs are a *FoldYielder*
- **pred_cb** (*PredHandler*) – *PredHandler* callback to determin how predictions are computed. Default simply returns the model predictions. Other uses could be e.g. running argmax on a multiclass classifier
- **cbs** (Optional[List[AbsCallback]]) – list of any instantiated callbacks to use during prediction
- **bs** (Optional[int]) – if not *None*, will run prediction in batches of specified size to save of memory

Return type Union[ndarray, Tensor, None]

Returns if inputs are a Numpy array, Pandas DataFrame, or tensor, will return predicitions as either array or tensor

save(*name*)
Save model, optimiser, and input mask states to file

Parameters **name** (str) – name of save file

Return type None

set_input_mask(*mask*)
Mask input columns by only using input columns whose indeces are listed in mask

Parameters **mask** (ndarray) – array of column indeces to use from all input columns

Return type None

set_lr(*lr*)
set learning rate of optimiser

Parameters **lr** (float) – learning rate of optimiser

Return type None

set_mom(*mom*)
Set momentum/beta_1 of optimiser

Parameters **mom** (float) – momentum/beta_1 of optimiser

Return type None

set_weights(*weights*)
Set state_dict of weights for network

Parameters **weights** (OrderedDict) – state_dict of weights for network

Return type None

unfreeze_layers()
Make parameters trainable

Return type None

lumin.nn.models.model_builder module

class lumin.nn.models.model_builder.**ModelBuilder**(*objective*, *n_out*, *cont_feats=None*, *model_args=None*, *opt_args=None*, *cat_embedder=None*, *cont_subsample_rate=None*, *guaranteed_feats=None*, *loss='auto'*, *head=<class 'lumin.nn.models.blocks.head.CatEmbHead'>*, *body=<class 'lumin.nn.models.blocks.body.FullyConnected'>*, *tail=<class 'lumin.nn.models.blocks.tail.ClassRegMulti'>*, *lookup_init=<function lookup_normal_init>*, *lookup_act=<function lookup_act>*, *pretrain_file=None*, *freeze_head=False*, *freeze_body=False*, *freeze_tail=False*)

Bases: object

Class to build models to specified architecture on demand along with an optimiser.

Parameters

- **objective** (str) – string representation of network objective, i.e. 'classification', 'regression', 'multiclass'
- **n_out** (int) – number of outputs required
- **cont_feats** (Optional[List[str]]) – list of names of continuous input features
- **model_args** (Optional[Dict[str, Dict[str, Any]]]) – dictionary of dictionaries of keyword arguments to pass to head, body, and tail to control architrcture
- **opt_args** (Optional[Dict[str, Any]]) – dictionary of arguments to pass to optimiser. Missing kargs will be filled with default values. Currently, only ADAM (default), and SGD are available.

- **cat_embedder** (Optional[*CatEmbedder*]) – *CatEmbedder* for embedding categorical inputs
- **cont_subsample_rate** (Optional[float]) – if between in range (0, 1), will randomly select a fraction of continuous features (rounded upwards) to use as inputs
- **guaranteed_feats** (Optional[List[str]]) – if subsampling features, will always include the features listed here, which count towards the subsample fraction
- **loss** (Any) – either and uninstantiated loss class, or leave as 'auto' to select loss according to objective
- **head** (Callable[[Any], AbsHead]) – uninstantiated class which can receive input data and upscale it to model width
- **body** (Callable[[Any], AbsBody]) – uninstantiated class which implements the main bulk of the model's hidden layers
- **tail** (Callable[[Any], AbsTail]) – uninstantiated class which scales the body to the required number of outputs and implements any final activation function and output scaling
- **lookup_init** (Callable[[str, Optional[int], Optional[int]], Callable[[Tensor], None]]) – function taking choice of activation function, number of inputs, and number of outputs an returning a function to initialise layer weights.
- **lookup_act** (Callable[[str], Module]) – function taking choice of activation function and returning an activation function layer
- **pretrain_file** (Optional[str]) – if set, will load saved parameters for entire network from saved model
- **freeze_head** (bool) – whether to start with the head parameters set to untrainable
- **freeze_body** (bool) – whether to start with the body parameters set to untrainable

Examples::

```
>>> model_builder = ModelBuilder(objective='classifier',
>>>                              cont_feats=cont_feats, n_out=1,
>>>                              model_args={'body':{'depth':4,
>>>                                                  'width':100}})
>>>
>>> min_targs = np.min(targets, axis=0).reshape(targets.shape[1],1)
>>> max_targs = np.max(targets, axis=0).reshape(targets.shape[1],1)
>>> min_targs[min_targs > 0] *=0.8
>>> min_targs[min_targs < 0] *=1.2
>>> max_targs[max_targs > 0] *=1.2
>>> max_targs[max_targs < 0] *=0.8
>>> y_range = np.hstack((min_targs, max_targs))
>>> model_builder = ModelBuilder(
>>>     objective='regression', cont_feats=cont_feats, n_out=6,
>>>     cat_embedder=CatEmbedder.from_fy(train_fy),
>>>     model_args={'body':{'depth':4, 'width':100},
>>>                 'tail':{y_range=y_range})
>>>
>>> model_builder = ModelBuilder(objective='multiclassifier',
>>>                              cont_feats=cont_feats, n_out=5,
>>>                              model_args={'body':{'width':100,
>>>                                                  'depth':6,
```

(continues on next page)

(continued from previous page)

```
>>>                                                 'do':0.1,
>>>                                                 'res':True}})
>>>
>>> model_builder = ModelBuilder(objective='classifier',
>>>                              cont_feats=cont_feats, n_out=1,
>>>                              model_args={'body':{'depth':4,
>>>                                                  'width':100}},
>>>                              opt_args={'opt':'sgd',
>>>                                        'momentum':0.8,
>>>                                        'weight_decay':1e-5},
>>>                              loss=partial(SignificanceLoss,
>>>                                           sig_weight=sig_weight,
>>>                                           bkg_weight=bkg_weight,
>>>                                           func=calc_ams_torch))
```

build_model()

Construct entire network module

Return type Module

Returns Instantiated nn.Module

classmethod from_model_builder(*model_builder*, *pretrain_file=None*, *freeze_head=False*, *freeze_body=False*, *freeze_tail=False*, *loss=None*, *opt_args=None*)

Instantiate a *ModelBuilder* from an exisitng *ModelBuilder*, but with options to adjust loss, optimiser, pretraining, and module freezing

Parameters

- **model_builder** – existing *ModelBuilder* or filename for a pickled *ModelBuilder*
- **pretrain_file** (Optional[str]) – if set, will load saved parameters for entire network from saved model
- **freeze_head** (bool) – whether to start with the head parameters set to untrainable
- **freeze_body** (bool) – whether to start with the body parameters set to untrainable
- **freeze_tail** (bool) – whether to start with the tail parameters set to untrainable
- **loss** (Optional[Any]) – either and uninstantiated loss class, or leave as 'auto' to select loss according to objective
- **opt_args** (Optional[Dict[str, Any]]) – dictionary of arguments to pass to optimiser. Missing kargs will be filled with default values. Choice of optimiser ('*opt*') keyword can either be set by passing the string name (e.g. *'adam'*), but only ADAM and SGD are available this way, or by passing an uninstantiated optimiser (e.g. torch.optim.Adam). If no optimser is set, then it defaults to ADAM. Additional keyword arguments can be set, and these will be passed tot he optimiser during instantiation

Returns Instantiated *ModelBuilder*

Examples::

```
>>> new_model_builder = ModelBuilder.from_model_builder(
>>>     ModelBuidler)
>>>
>>> new_model_builder = ModelBuilder.from_model_builder(
>>>     ModelBuidler, loss=partial(
>>>         SignificanceLoss, sig_weight=sig_weight,
>>>         bkg_weight=bkg_weight, func=calc_ams_torch))
>>>
>>> new_model_builder = ModelBuilder.from_model_builder(
>>>     'weights/model_builder.pkl',
>>>     opt_args={'opt':'sgd', 'momentum':0.8, 'weight_decay':1e-5})
>>>
>>> new_model_builder = ModelBuilder.from_model_builder(
>>>     'weights/model_builder.pkl',
>>>     opt_args={'opt':torch.optim.Adam,
...               'momentum':0.8,
...               'weight_decay':1e-5})
```

get_body(*n_in*, *feat_map*)
Construct body module

Return type AbsBody

Returns Instantiated body nn.Module

get_head()
Construct head module

Return type AbsHead

Returns Instantiated head nn.Module

get_model()
Construct model, loss, and optimiser, optionally loading pretrained weights

Return type Tuple[Module, Optimizer, Callable[[], Module], Optional[ndarray]]

Returns Instantiated network, optimiser linked to model parameters, uninstantiated loss, and optional input mask

get_out_size()
Get number of outputs of model

Return type int

Returns number of outputs of network

get_tail(*n_in*)
Construct tail module

Return type Module

Returns Instantiated tail nn.Module

load_pretrained(*model*)
Load model weights from pretrained file

Parameters **model** (Module) – instantiated model, i.e. return of *build_model()*

Returns model with weights loaded

set_lr(*lr*)
Set learning rate for all model parameters

Parameters **lr** (float) – learning rate

Return type None

Module contents

6.1.8 lumin.nn.training package

Submodules

lumin.nn.training.train module

lumin.nn.training.train.**train_models**(*fy*, *n_models*, *bs*, *model_builder*, *n_epochs*, *patience=None*, *loss_is_meaned=True*, *cb_partials=None*, *metric_partials=None*, *save_best=True*, *train_on_weights=True*, *bulk_move=True*, *start_model_id=0*, *excl_idxs=None*, *unique_trn_idxs=False*, *live_fdbk=False*, *live_fdbk_first_only=False*, *live_fdbk_extra=True*, *live_fdbk_extra_first_only=False*, *savepath=Path('train_weights')*, *plot_settings=<lumin.plotting.plot_settings.PlotSettings object>*)

Main training method for *Model*. Trains a specified numer of models created by a *ModelBuilder* on data provided by a *FoldYielder*, and saves them to *savepath*.

Note, this does not return trained models, instead they are saved and must be loaded later. Instead this method returns results of model training. Each *Model* is trained on N-1 folds, for a *FoldYielder* with N folds, and the remaining fold is used as validation data.

Depending on the live_fdbk arguments, live plots of losses and other metrics may be shown during training, if running in Jupyter. Showing the live plot slightly slows down the training, but can help highlight problems without having to wait to the end. If not running in Jupyter, then losses are printed to the terminal.

Once training is finished, the state with the lowest validation loss is loaded, evaluated, and saved.

Parameters

- **fy** (*FoldYielder*) – *FoldYielder* interfacing ot training data
- **n_models** (int) – number of models to train
- **bs** (int) – batch size. Number of data points per iteration
- **model_builder** (*ModelBuilder*) – *ModelBuilder* creating the networks to train
- **n_epochs** (int) – maximum number of epochs for which to train
- **patience** (Optional[int]) – if not *None*, sets the number of epochs or cycles to train without decrease in validation loss before ending training (early stopping)
- **loss_is_meaned** (bool) – if the batch loss value has been averaged over the number of elements in the batch, this should be true
- **cb_partials** (Optional[List[Callable[[], *Callback*]]]) – optional list of functools.partial, each of which will instantiate a *Callback* when called

- **metric_partials** (Optional[List[Callable[[], *EvalMetric*]]]) – optional list of functools.partial, each of which will a instantiate EvalMetric, used to compute additional metrics on validation data after each epoch. *SaveBest* and *EarlyStopping* will also act on the (first) metric set to *main_metric* instead of loss, except when another callback produces an alternative loss and model (like *SWA*).
- **save_best** (bool) – if true, will save the best performing model as the final model, otherwise will save the model state as per the end of training. A copy of the best model will still be saved anyway.
- **train_on_weights** (bool) – If weights are present in training data, whether to pass them to the loss function during training
- **bulk_move** (bool) – if true, will optimise for speed by using more RAM and VRAM
- **start_model_id** (int) – model ID at whcih to start training, i.e. if training was interupted, this can be set to resume training form the last model which was trained
- **excl_idxs** (Optional[List[int]]) – optional list of fold indeces to exclude from training and validation
- **unique_trn_idxs** (bool) – if false, then fold indeces can be shared, e.g. if *fy* contains 10 folds and five models are requested, each model will be trained on 9 folds. if true, each model will every model will be trained on different folds, e.g. if *fy* contains 10 folds and five models are requested, each model will be trained on 2 folds and no same fold is used to train more than one model This is useful when the amount of training data exceeds the amount required to train a single model: it can be split into a large number of folds and a set of decorellated models trained.
- **live_fdbk** (bool) – whether or not to show any live feedback at all during training (slightly slows down training, but helps spot problems)
- **live_fdbk_first_only** (bool) – whether to only show live feedback for the first model trained (trade off between time and problem spotting)
- **live_fdbk_extra** (bool) – whether to show extra information live feedback (further slows training)
- **live_fdbk_extra_first_only** (bool) – whether to only show extra live feedback information for the first model trained (trade off between time and information)
- **savepath** (Path) – path to to which to save model weights and results
- **plot_settings** (*PlotSettings*) – *PlotSettings* class to control figure appearance

Return type Tuple[List[Dict[str, float]], List[Dict[str, List[float]]], List[Dict[str, float]]]

Returns

- results list of validation losses and other eval_metrics results, ordered by model training. Can be used to create an *Ensemble*.
- histories list of loss histories, ordered by model training
- cycle_losses if an *AbsCyclicCallback* was passed, lists validation losses at the end of each cycle, ordered by model training. Can be passed to *Ensemble*.

6.2 Module contents

CHAPTER

SEVEN

LUMIN.OPTIMISATION PACKAGE

7.1 Submodules

7.2 lumin.optimisation.features module

lumin.optimisation.features.**get_rf_feat_importance**(*rf*, *inputs*, *targets*, *weights=None*)

Compute feature importance for a Random Forest model using rfpimp.

Parameters

- **rf** (Union[RandomForestRegressor, RandomForestClassifier]) – trained Random Forest model
- **inputs** (DataFrame) – input data as Pandas DataFrame
- **targets** (ndarray) – target data as Numpy array
- **weights** (Optional[ndarray]) – Optional data weights as Numpy array

Return type DataFrame

lumin.optimisation.features.**rf_rank_features**(*train_df*, *val_df*, *objective*, *train_feats*, *targ_name='gen_target'*, *wgt_name=None*, *importance_cut=0.0*, *n_estimators=40*, *rf_params=None*, *optimise_rf=True*, *n_rfs=1*, *n_max_display=30*, *plot_results=True*, *retrain_on_import_feats=True*, *verbose=True*, *savename=None*, *plot_settings=<lumin.plotting.plot_settings.PlotSettings object>*)

Compute relative permutation importance of input features via using Random Forests. A reduced set of 'important features' is obtained by cutting on relative importance and a new model is trained and evaluated on this reduced set. RFs will have their hyper-parameters roughly optimised, both when training on all features and once when training on important features. Relative importances may be computed multiple times (via n_rfs) and averaged. In which case the standard error is also computed.

Parameters

- **train_df** (DataFrame) – training data as Pandas DataFrame
- **val_df** (DataFrame) – validation data as Pandas DataFrame
- **objective** (str) – string representation of objective: either 'classification' or 'regression'
- **train_feats** (List[str]) – complete list of training features

- **targ_name** (str) – name of column containing target data
- **wgt_name** (Optional[str]) – name of column containing weight data. If set, will use weights for training and evaluation, otherwise will not
- **importance_cut** (float) – minimum importance required to be considered an 'important feature'
- **n_estimators** (int) – number of trees to use in each forest
- **rf_params** (Optional[Dict[str, Any]]) – optional dictionary of keyword parameters for SK-Learn Random Forests Or ordered dictionary mapping parameters to optimise to list of values to consider If None and will optimise parameters using *lumin.optimisation.hyper_param.get_opt_rf_params()*
- **optimise_rf** (bool) – if true will optimise RF params, passing *rf_params* to *get_opt_rf_params()*
- **n_rfs** (int) – number of trainings to perform on all training features in order to compute importances
- **n_max_display** (int) – maximum number of features to display in importance plot
- **plot_results** (bool) – whether to plot the feature importances
- **retrain_on_import_feats** (bool) – whether to train a new model on important features to compare to full model
- **verbose** (bool) – whether to report results and progress
- **savename** (Optional[str]) – Optional name of file to which to save the plot of feature importances
- **plot_settings** (*PlotSettings*) – *PlotSettings* class to control figure appearance

Return type List[str]

Returns List of features passing importance_cut, ordered by decreasing importance

lumin.optimisation.features.**rf_check_feat_removal**(*train_df*, *objective*, *train_feats*, *check_feats*, *targ_name='gen_target'*, *wgt_name=None*, *val_df=None*, *subsample_rate=None*, *strat_key=None*, *n_estimators=40*, *n_rfs=1*, *rf_params=None*)

Checks whether features can be removed from the set of training features without degrading model performance using Random Forests Computes scores for model with all training features then for each feature listed in *check_feats* computes scores for a model trained on all training features except that feature E.g. if two features are highly correlated this function could be used to check whether one of them could be removed.

Parameters

- **train_df** (DataFrame) – training data as Pandas DataFrame
- **objective** (str) – string representation of objective: either 'classification' or 'regression'
- **train_feats** (List[str]) – complete list of training features
- **check_feats** (List[str]) – list of features to try removing
- **targ_name** (str) – name of column containing target data

- **wgt_name** (Optional[str]) – name of column containing weight data. If set, will use weights for training and evaluation, otherwise will not
- **val_df** (Optional[DataFrame]) – optional validation data as Pandas DataFrame. If set will compute validation scores in addition to Out Of Bag scores And will optimise RF parameters if *rf_params* is None
- **subsample_rate** (Optional[float]) – if set, will subsample the training data to the provided fraction. Subsample is repeated per Random Forest training
- **strat_key** (Optional[str]) – column name to use for stratified subsampling, if desired
- **n_estimators** (int) – number of trees to use in each forest
- **n_rfs** (int) – number of trainings to perform on all training features in order to compute importances
- **rf_params** (Optional[Dict[str, Any]]) – optional dictionary of keyword parameters for SK-Learn Random Forests If None and val_df is None will use default parameters of 'min_samples_leaf':3, 'max_features':0.5 Elif None and val_df is not None will optimise parameters using *lumin.optimisation.hyper_param.get_opt_rf_params()*

Return type Dict[str, float]

Returns Dictionary of results

lumin.optimisation.features.**repeated_rf_rank_features**(*train_df*, *val_df*, *n_reps*, *min_frac_import*, *objective*, *train_feats*, *targ_name='gen_target'*, *wgt_name=None*, *strat_key=None*, *subsample_rate=None*, *resample_val=True*, *importance_cut=0.0*, *n_estimators=40*, *rf_params=None*, *optimise_rf=True*, *n_rfs=1*, *n_max_display=30*, *n_threads=1*, *savename=None*, *plot_settings=<lumin.plotting.plot_settings.PlotSettin*
object>)

Runs *rf_rank_features()* multiple times on bootstrap resamples of training data and computes the fraction of times each feature passes the importance cut. Then returns a list features which are have a fractional selection as important great than some number. I.e. in cases where *rf_rank_features()* can be unstable (list of important features changes each run), this method can be used to help stabailse the list of important features

Parameters

- **train_df** (DataFrame) – training data as Pandas DataFrame
- **val_df** (DataFrame) – validation data as Pandas DataFrame
- **n_reps** (int) – number of times to resample and run *rf_rank_features()*
- **min_frac_import** (float) – minimum fraction of times feature must be selected as important by *rf_rank_features()* in order to be considered generally important

- **objective** (str) – string representation of objective: either 'classification' or 'regression'
- **train_feats** (List[str]) – complete list of training features
- **targ_name** (str) – name of column containing target data
- **wgt_name** (Optional[str]) – name of column containing weight data. If set, will use weights for training and evaluation, otherwise will not
- **strat_key** (Optional[str]) – name of column to use to stratify data when resampling
- **subsample_rate** (Optional[float]) – if set, will subsample the training data to the provided fraction. Subsample is repeated per Random Forest training
- **resample_val** (bool) – whether to also resample the validation set, or use the original set for all evaluations
- **importance_cut** (float) – minimum importance required to be considered an 'important feature'
- **n_estimators** (int) – number of trees to use in each forest
- **rf_params** (Optional[Dict[str, Any]]) – optional dictionary of keyword parameters for SK-Learn Random Forests Or ordered dictionary mapping parameters to optimise to list of values to consider If None and will optimise parameters using *lumin.optimisation.hyper_param.get_opt_rf_params()*
- **optimise_rf** (bool) – if true will optimise RF params, passing *rf_params* to *get_opt_rf_params()*
- **n_rfs** (int) – number of trainings to perform on all training features in order to compute importances
- **n_max_display** (int) – maximum number of features to display in importance plot
- **n_threads** (int) – number of rankings to run simultaneously
- **savename** (Optional[str]) – Optional name of file to which to save the plot of feature importances
- **plot_settings** (*PlotSettings*) – *PlotSettings* class to control figure appearance

Return type Tuple[List[str], DataFrame]

Returns

- List of features with fractional selection greater than min_frac_import, ordered by decreasing fractional selection
- DataFrame of number of selections and fractional selections for all features

lumin.optimisation.features.**auto_filter_on_linear_correlation**(*train_df*, *val_df*, *check_feats*, *objective*, *targ_name*, *strat_key=None*, *wgt_name=None*, *corr_threshold=0.8*, *n_estimators=40*, *rf_params=None*, *optimise_rf=True*, *n_rfs=5*, *subsample_rate=None*, *savename=None*, *plot_settings=<lumin.plotting.plot_setting object>*)

Filters a list of possible training features by identifying pairs of linearly correlated features and then attempting to remove either feature from each pair by checking whether doing so would not decrease the performance Random Forests trained to perform classification or regression.

Linearly correlated features are identified by computing Spearman's rank-order correlation coefficients for every pair of features. Hierachical clustering is then used to group features. Clusters of features with a correlation coefficient greater than a set threshold are candidates for removal. Candidate sets of features are tested, in order of decreasing correlation, by computing the mean performance of a Random Forests trained on all remaining training features and all remaining training features except each feature in the set in turn. If the RF trained on all remaining features consistently outperforms the other trainings, then no feature from the set is removed, otherwise the feature whose removal causes the largest mean increase in performance is removed. This test is then repeated on the remaining features in the set, until either no features are removed, or only one feature remains.

Since this function involves training many models, it can be slow on large datasets. In such cases one can use the *subsample_rate* argument to sample randomly a fraction of the whole dataset (with optionaly stratification). Resampling is performed prior to each RF training for maximum genralisation, and any weights in the data are automatically renormalised to the original weight sum (within each class).

Attention: This function combines *plot_rank_order_dendrogram()* with *rf_check_feat_removal()*. This is purely for convenience and should not be treated as a 'black box'. We encourage users to convince themselves that it is really is reasonable to remove the features which are identified as redundant.

Parameters

- **train_df** (DataFrame) – training data as Pandas DataFrame
- **val_df** (DataFrame) – validation data as Pandas DataFrame
- **check_feats** (List[str]) – complete list of features to consider for training and removal
- **objective** (str) – string representation of objective: either 'classification' or 'regression'
- **targ_name** (str) – name of column containing target data
- **strat_key** (Optional[str]) – name of column to use to stratify data when resampling

- **wgt_name** (Optional[str]) – name of column containing weight data. If set, will use weights for training and evaluation, otherwise will not
- **corr_threshold** (float) – minimum threshold on Spearman's rank-order correlation coefficient for pairs to be considered 'correlated'
- **n_estimators** (int) – number of trees to use in each forest
- **rf_params** (Optional[Dict[~KT, ~VT]]) – either: a dictionare of keyword hyper-parameters to use for the Random Forests, if optimse_rf is False; or an *OrderedDict* of a range of hyper-parameters to test during optimisation. See *get_opt_rf_params()* for more details.
- **optimise_rf** (bool) – whether to optimise the Random Forest hyper-parameters for the (sub-sambled) dataset
- **n_rfs** (int) – number of trainings to perform during each perfromance impact test
- **subsample_rate** (Optional[float]) – float between 0 and 1. If set will sub-sample the trainng data to the requested fraction
- **savename** (Optional[str]) – Optional name of file to which to save the first plot of feature clustering
- **plot_settings** (*PlotSettings*) – *PlotSettings* class to control figure appearance

Return type List[str]

Returns Filtered list of training features

lumin.optimisation.features.**auto_filter_on_mutual_dependence**(*train_df*, *val_df*, *check_feats*, *objective*, *targ_name*, *strat_key=None*, *wgt_name=None*, *md_threshold=0.8*, *n_estimators=40*, *rf_params=None*, *optimise_rf=True*, *n_rfs=5*, *subsample_rate=None*, *plot_settings=<lumin.plotting.plot_settings.object>*)

Filters a list of possible training features via mutual dependence: By identifying features whose values can be accurately predicted using the other features. Features with a high 'dependence' are then checked to see whether removing them would not decrease the performance Random Forests trained to perform classification or regression. For best results, the features to check should be supplied in order to decreasing importance.

Dependent features are identified by training Random Forest regressors on the other features. Features with a dependence greater than a set threshold are candidates for removal. Candidate features are tested, in order of increasing importance, by computing the mean performance of a Random Forests trained on: all remaining training features; and all remaining training features except the candidate feature. If the RF trained on all remaining features except the candidate feature consistently outperforms or matches the training which uses all remaining features, then the candidate feature is removed, otherwise the feature remains and is no longer tested.

Since evaluating the mutual dependence via regression then allows the important features used by the regressor to be identified, it is possible to test multiple feature removals at once, provided a removal candidate is not important for predicting another removal candidate.

Since this function involves training many models, it can be slow on large datasets. In such cases one can use the *subsample_rate* argument to sample randomly a fraction of the whole dataset (with optionaly stratification). Resampling is performed prior to each RF training for maximum genralisation, and any weights in the data are automatically renormalised to the original weight sum (within each class).

Attention: This function combines RFPImp's *feature_dependence_matrix* with *rf_check_feat_removal()*. This is purely for convenience and should not be treated as a 'black box'. We encourage users to convince themselves that it is really is reasonable to remove the features which are identified as redundant.

Note: Technicalities related to RFPImp's use of SVG for plots mean that the mutual dependence plots can have low resolution when shown or saved. Therefore this function does not take a *savename* argument. Users wiching to save the plots as PNG or PDF should compute the dependence matrix themselves using *feature_dependence_matrix* and then plot using *plot_dependence_heatmap*, calling *.save([savename])* on the returned object. The plotting backend might need to be set to SVG, using: *%config InlineBackend.figure_format = 'svg'*.

Parameters

- **train_df** (DataFrame) – training data as Pandas DataFrame
- **val_df** (DataFrame) – validation data as Pandas DataFrame
- **check_feats** (List[str]) – complete list of features to consider for training and removal
- **objective** (str) – string representation of objective: either 'classification' or 'regression'
- **targ_name** (str) – name of column containing target data
- **strat_key** (Optional[str]) – name of column to use to stratify data when resampling
- **wgt_name** (Optional[str]) – name of column containing weight data. If set, will use weights for training and evaluation, otherwise will not
- **md_threshold** (float) – minimum threshold on the mutual dependence coefficient for a feature to be considered 'predictable'
- **n_estimators** (int) – number of trees to use in each forest
- **rf_params** (Optional[OrderedDict]) – either: a dictionare of keyword hyper-parameters to use for the Random Forests, if optimse_rf is False; or an *OrderedDict* of a range of hyper-parameters to test during optimisation. See *get_opt_rf_params()* for more details.
- **optimise_rf** (bool) – whether to optimise the Random Forest hyper-parameters for the (sub-sambled) dataset
- **n_rfs** (int) – number of trainings to perform during each perfromance impact test
- **subsample_rate** (Optional[float]) – float between 0 and 1. If set will subsample the trainng data to the requested fraction
- **plot_settings** (*PlotSettings*) – *PlotSettings* class to control figure appearance

Return type List[str]

Returns Filtered list of training features

7.3 lumin.optimisation.hyper_param module

lumin.optimisation.hyper_param.**get_opt_rf_params**(*x_trn*, *y_trn*, *x_val*, *y_val*, *objective*, *w_trn=None*, *w_val=None*, *params=None*, *n_estimators=40*, *verbose=True*)

Use an ordered parameter-scan to roughly optimise Random Forest hyper-parameters.

Parameters

- **x_trn** (ndarray) – training input data
- **y_trn** (ndarray) – training target data
- **x_val** (ndarray) – validation input data
- **y_val** (ndarray) – validation target data
- **objective** (str) – string representation of objective: either 'classification' or 'regression'
- **w_trn** (Optional[ndarray]) – training weights
- **w_val** (Optional[ndarray]) – validation weights
- **params** (Optional[OrderedDict]) – ordered dictionary mapping parameters to optimise to list of values to cosnider
- **n_estimators** (int) – number of trees to use in each forest
- **verbose** – Print extra information and show a live plot of model performance

Returns dictionary mapping parameters to their optimised values rf: best performing Random Forest

Return type params

lumin.optimisation.hyper_param.**lr_find**(*fy*, *model_builder*, *bs*, *n_epochs=1*, *train_on_weights=True*, *n_repeats=-1*, *lr_bounds=[1e-05, 10]*, *cb_partials=None*, *plot_settings=<lumin.plotting.plot_settings.PlotSettings object>*, *bulk_move=True*, *plot_savename=None*)

Wrapper function for training using *LRFinder* which runs a Smith LR range test (https://arxiv.org/abs/1803.09820) using folds in *FoldYielder*. Trains models for a set number of repeats, interpolating LR between set bounds. This repeats for each fold in *FoldYielder*, and loss evolution is averaged.

Parameters

- **fy** (*FoldYielder*) – *FoldYielder* providing training data
- **model_builder** (*ModelBuilder*) – *ModelBuilder* providing networks and optimisers
- **bs** (int) – batch size
- **n_epochs** (int) – number of epochs to train per fold
- **train_on_weights** (bool) – If weights are present, whether to use them for training
- **shuffle_fold** – whether to shuffle data in folds

- **n_folds** – if >= 1, will only train n_folds number of models, otherwise will train one model per fold
- **lr_bounds** (Tuple[float, float]) – starting and ending LR values
- **cb_partials** (Optional[List[partial]]) – optional list of functools.partial, each of which will a instantiate *Callback* when called
- **plot_settings** (*PlotSettings*) – *PlotSettings* class to control figure appearance
- **savename** – Optional name of file to which to save the plot

Return type List[*LRFinder*]

Returns List of *LRFinder* which were used for each model trained

7.4 lumin.optimisation.threshold module

lumin.optimisation.threshold.**binary_class_cut_by_ams**(*df*, *top_perc=5.0*, *min_pred=0.9*, *wgt_factor=1.0*, *br=0.0*, *syst_unc_b=0.0*, *pred_name='pred'*, *targ_name='gen_target'*, *wgt_name='gen_weight'*, *plot_settings=<lumin.plotting.plot_settings.PlotSettings object>*)

Optimise a cut on a signal-background classifier prediction by the Approximate Median Significance Cut which should generalise better by taking the mean class prediction of the top top_perc percentage of points as ranked by AMS

Parameters

- **df** (DataFrame) – Pandas DataFrame containing data
- **top_perc** (float) – top percentage of events to consider as ranked by AMS
- **min_pred** (float) – minimum prediction to consider
- **wgt_factor** (float) – single multiplicative coeficient for rescaling signal and background weights before computing AMS
- **br** (float) – background offset bias
- **syst_unc_b** (float) – fractional systemtatic uncertainty on background
- **pred_name** (str) – column to use as predictions
- **targ_name** (str) – column to use as truth labels for signal and background
- **wgt_name** (str) – column to use as weights for signal and background events
- **plot_settings** (*PlotSettings*) – *PlotSettings* class to control figure appearance

Return type Tuple[float, float, float]

Returns Optimised cut AMS at cut Maximum AMS

7.5 Module contents

CHAPTER

EIGHT

LUMIN.PLOTTING PACKAGE

8.1 Submodules

8.2 lumin.plotting.data_viewing module

lumin.plotting.data_viewing.**plot_feat**(*df*, *feat*, *wgt_name=None*, *cuts=None*, *labels=''*, *plot_bulk=True*, *n_samples=100000*, *plot_params=None*, *size='mid'*, *show_moments=True*, *ax_labels={'x': None, 'y': None}*, *log_x=False*, *log_y=False*, *savename=None*, *settings=<lumin.plotting.plot_settings.PlotSettings object>*)

A flexible function to provide indicative information about the 1D distribution of a feature. By default it will produce a weighted KDE+histogram for the [1,99] percentile of the data, as well as compute the mean and standard deviation of the data in this region. Distributions are weighted by sampling with replacement the data with probabilities propotional to the sample weights. By passing a list of cuts and labels, it will plot multiple distributions of the same feature for different cuts. Since it is designed to provide quick, indicative information, more specific functions (such as *plot_kdes_from_bs*) should be used to provide final results.

Important: NaN and Inf values are removed prior to plotting and no attempt is made to coerce them to real numbers

---

Parameters

- **df** (DataFrame) – Pandas DataFrame containing data
- **feat** (str) – column name to plot
- **wgt_name** (Optional[str]) – if set, will use column to weight data
- **cuts** (Optional[List[Series]]) – optional list of cuts to apply to feature. Will add one KDE+hist for each cut listed on the same plot
- **labels** (Optional[List[str]]) – optional list of labels for each KDE+hist
- **plot_bulk** (bool) – whether to plot the [1,99] percentile of the data, or all of it
- **n_samples** (int) – if plotting weighted distributions, how many samples to use
- **plot_params** (Union[Dict[str, Any], List[Dict[str, Any]], None]) – optional list of of arguments to pass to Seaborn Distplot for each KDE+hist
- **size** (str) – string to pass to *str2sz()* to determin size of plot

- **show_moments** (bool) – whether to compute and display the mean and standard deviation
- **ax_labels** (Dict[str, Any]) – dictionary of x and y axes labels
- **log_x** (bool) – if true, will use log scale for x-axis
- **log_y** (bool) – if true, will use log scale for y-axis
- **savename** (Optional[str]) – Optional name of file to which to save the plot of feature importances
- **settings** (*PlotSettings*) – *PlotSettings* class to control figure appearance

Return type None

lumin.plotting.data_viewing.**compare_events**(*events*)

Plots at least two events side by side in their transverse and longitudinal projections

Parameters **events** (list) – list of DataFrames containing vector coordinates for 3 momenta

Return type None

lumin.plotting.data_viewing.**plot_rank_order_dendrogram**(*df*, *threshold=0.8*, *savename=None*, *settings=<lumin.plotting.plot_settings.PlotSettings object>*)

Plots a dendrogram of features in df clustered via Spearman's rank correlation coefficient. Also returns a sets of features with correlation coefficients greater than the threshold

Parameters

- **df** (DataFrame) – Pandas DataFrame containing data
- **threshold** (float) – Threshold on correlation coefficient
- **savename** (Optional[str]) – Optional name of file to which to save the plot of feature importances
- **settings** (*PlotSettings*) – *PlotSettings* class to control figure appearance

Return type Dict[str, Union[List[str], float]]

Returns Dict of sets of features with correlation coefficients greater than the threshold and cluster distance

lumin.plotting.data_viewing.**plot_kdes_from_bs**(*x*, *bs_stats*, *name2args*, *feat*, *units=None*, *moments=True*, *savename=None*, *settings=<lumin.plotting.plot_settings.PlotSettings object>*)

Plots KDEs computed via *bootstrap_stats()*

Parameters

- **bs_stats** (Dict[str, Any]) – (filtered) dictionary retruned by *bootstrap_stats()*
- **name2args** (Dict[str, Dict[str, Any]]) – Dictionary mapping names of different distributions to arguments to pass to seaborn tsplot
- **feat** (str) – Name of feature being plotted (for axis lablels)
- **units** (Optional[str]) – Optional units to show on axes
- **moments** – whether to display mean and standard deviation of each distribution
- **savename** (Optional[str]) – Optional name of file to which to save the plot of feature importances

- **settings** (*PlotSettings*) – *PlotSettings* class to control figure appearance

Return type None

lumin.plotting.data_viewing.**plot_binary_sample_feat**(*df*, *feat*, *targ_name='gen_target'*, *wgt_name='gen_weight'*, *sample_name='gen_sample'*, *wgt_scale=1*, *bins=None*, *log_y=False*, *lim_x=None*, *density=True*, *feat_name=None*, *units=None*, *savename=None*, *settings=<lumin.plotting.plot_settings.PlotSettings object>*)

More advanced plotter for feature distributions in a binary class problem with stacked distributions for backgrounds and user-defined binning Note that plotting colours can be controled by seeting the settings.sample2col dictionary

Parameters

- **df** (DataFrame) – DataFrame with targets and predictions
- **feat** (str) – name of column to plot the distribution of
- **targ_name** (str) – name of column to use as targets
- **wgt_name** (str) – name of column to use as sample weights
- **sample_name** (str) – name of column to use as process names
- **wgt_scale** (float) – applies a global multiplicative rescaling to sample weights. Default 1 = no rescaling. Only applicable when density = False
- **bins** (Union[int, List[int], None]) – either the number of bins to use for a uniform binning, or a list of bin edges for a variable-width binning
- **log_y** (bool) – whether to use a log scale for the y-axis
- **lim_x** (Optional[Tuple[float, float]]) – limit for plotting on the x-axis
- **density** – whether to normalise each distribution to one, or keep set to sum of weights / datapoints
- **feat_name** (Optional[str]) – Name of feature to put on x-axis, can be in LaTeX.
- **units** (Optional[str]) – units used to measure feature, if applicable. Can be in LaTeX, but should not include '$'.
- **savename** (Optional[str]) – Optional name of file to which to save the plot of feature importances
- **settings** (*PlotSettings*) – *PlotSettings* class to control figure appearance

Return type None

8.3 lumin.plotting.interpretation module

lumin.plotting.interpretation.**plot_importance**(*df*, *feat_name='Feature'*, *imp_name='Importance'*, *unc_name='Uncertainty'*, *threshold=None*, *x_lbl='Importance via feature permutation'*, *savename=None*, *settings=<lumin.plotting.plot_settings.PlotSettings object>*)

Plot feature importances as computted via *get_nn_feat_importance*, *get_ensemble_feat_importance*, or *rf_rank_features*

Parameters

- **df** (DataFrame) – DataFrame containing columns of features, importances and, optionally, uncertainties
- **feat_name** (str) – column name for features
- **imp_name** (str) – column name for importances
- **unc_name** (str) – column name for uncertainties (if present)
- **threshold** (Optional[float]) – if set, will draw a line at the threshold hold used for feature importance
- **x_lbl** (str) – label to put on the x-axis
- **savename** (Optional[str]) – Optional name of file to which to save the plot of feature importances
- **settings** (*PlotSettings*) – *PlotSettings* class to control figure appearance

Return type None

lumin.plotting.interpretation.**plot_embedding**(*embed*, *feat*, *savename=None*, *settings=<lumin.plotting.plot_settings.PlotSettings object>*)

Visualise weights in provided categorical entity-embedding matrix

Parameters

- **embed** (OrderedDict) – state_dict of trained nn.Embedding
- **feat** (str) – name of feature embedded
- **savename** (Optional[str]) – Optional name of file to which to save the plot of feature importances
- **settings** (*PlotSettings*) – *PlotSettings* class to control figure appearance

Return type None

lumin.plotting.interpretation.**plot_1d_partial_dependence**(*model*, *df*, *feat*, *train_feats*, *ignore_feats=None*, *input_pipe=None*, *sample_sz=None*, *wgt_name=None*, *n_clusters=10*, *n_points=20*, *pdp_isolate_kargs=None*, *pdp_plot_kargs=None*, *y_lim=None*, *savename=None*, *settings=<lumin.plotting.plot_settings.PlotSettings object>*)

Wrapper for PDPbox to plot 1D dependence of specified feature using provided NN or RF. If features have been preprocessed using an SK-Learn Pipeline, then that can be passed in order to rescale the x-axis back to its original values.

Parameters

- **model** (Any) – any trained model with a .predict method
- **df** (DataFrame) – DataFrame containing training data
- **feat** (str) – feature for which to evaluate the partial dependence of the model
- **train_feats** (List[str]) – list of all training features including ones which were later ignored, i.e. input features considered when input_pipe was fitted
- **ignore_feats** (Optional[List[str]]) – features present in training data which were not used to train the model (necessary to correctly deprocess feature using input_pipe)
- **input_pipe** (Optional[Pipeline]) – SK-Learn Pipeline which was used to process the training data
- **sample_sz** (Optional[int]) – if set, will only compute partial dependence on a random sample with replacement of the training data, sampled according to weights (if set). Speeds up computation and allows weighted partial dependencies to computed.
- **wgt_name** (Optional[str]) – Optional column name to use as sampling weights
- **n_points** (int) – number of points at which to evaluate the model output, passed to pdp_isolate as num_grid_points
- **n_clusters** (Optional[int]) – number of clusters in which to group dependency lines. Set to None to show all lines
- **pdp_isolate_kargs** (Optional[Dict[str, Any]]) – optional dictionary of keyword arguments to pass to pdp_isolate
- **pdp_plot_kargs** (Optional[Dict[str, Any]]) – optional dictionary of keyword arguments to pass to pdp_plot
- **y_lim** (Union[Tuple[float, float], List[float], None]) – If set, will limit y-axis plot range to tuple
- **savename** (Optional[str]) – Optional name of file to which to save the plot of feature importances
- **settings** (*PlotSettings*) – *PlotSettings* class to control figure appearance

Return type None

lumin.plotting.interpretation.**plot_2d_partial_dependence**(*model, df, feats, train_feats, ignore_feats=None, input_pipe=None, sample_sz=None, wgt_name=None, n_points=[20, 20], pdp_interact_kargs=None, pdp_interact_plot_kargs=None, savename=None, settings=<lumin.plotting.plot_settings.PlotSettings object>*)

Wrapper for PDPbox to plot 2D dependence of specified pair of features using provided NN or RF. If features have been preprocessed using an SK-Learn Pipeline, then that can be passed in order to rescale them back to their original values.

Parameters

- **model** (Any) – any trained model with a .predict method
- **df** (DataFrame) – DataFrame containing training data
- **feats** (Tuple[str, str]) – pair of features for which to evaluate the partial dependence of the model
- **train_feats** (List[str]) – list of all training features including ones which were later ignored, i.e. input features considered when input_pipe was fitted
- **ignore_feats** (Optional[List[str]]) – features present in training data which were not used to train the model (necessary to correctly deprocess feature using input_pipe)
- **input_pipe** (Optional[Pipeline]) – SK-Learn Pipeline which was used to process the training data
- **sample_sz** (Optional[int]) – if set, will only compute partial dependence on a random sample with replacement of the training data, sampled according to weights (if set). Speeds up computation and allows weighted partial dependencies to computed.
- **wgt_name** (Optional[str]) – Optional column name to use as sampling weights
- **n_points** (Tuple[int, int]) – pair of numbers of points at which to evaluate the model output, passed to pdp_interact as num_grid_points
- **n_clusters** – number of clusters in which to group dependency lines. Set to None to show all lines
- **pdp_isolate_kargs** – optional dictionary of keyword arguments to pass to pdp_isolate
- **pdp_plot_kargs** – optional dictionary of keyword arguments to pass to pdp_plot
- **savename** (Optional[str]) – Optional name of file to which to save the plot of feature importances
- **settings** (*PlotSettings*) – *PlotSettings* class to control figure appearance

Return type None

lumin.plotting.interpretation.**plot_multibody_weighted_outputs**(*model*, *inputs*, *block_names=None*, *use_mean=False*, *savename=None*, *settings=<lumin.plotting.plot_settings.PlotSe object>*)

Interpret how a model relies on the outputs of each block in a :class:MultiBlock by plotting the outputs of each block as weighted by the tail block. This function currently only supports models whose tail block contains a single neuron in the first dense layer. Input data is passed through the model and the absolute sums of the weighted block outputs are computed per datum, and optionally averaged over the number of block outputs.

Parameters

- **model** (AbsModel) – model to interpret
- **inputs** (Union[ndarray, Tensor]) – input data to use for interpretation
- **block_names** (Optional[List[str]]) – names for each block to use when plotting
- **use_mean** (bool) – if True, will average the weighted outputs over the number of output neurons in each block
- **savename** (Optional[str]) – Optional name of file to which to save the plot of feature importances
- **settings** (*PlotSettings*) – *PlotSettings* class to control figure appearance

Return type None

lumin.plotting.interpretation.**plot_bottleneck_weighted_inputs**(*model*, *bottleneck_idx*, *inputs*, *log_y=True*, *savename=None*, *settings=<lumin.plotting.plot_settings.PlotSe object>*)

Interpret how a single-neuron bottleneck in a :class:MultiBlock relies on input features by plotting the absolute values of the features times their associated weight for a given set of input data.

Parameters

- **model** (AbsModel) – model to interpret
- **bottleneck_idx** (int) – index of the bottleneck to interpret, i.e. model.body.bottleneck_blocks[bottleneck_idx]
- **inputs** (Union[ndarray, Tensor]) – input data to use for interpretation
- **log_y** (bool) – whether to plot a log scale for the y-axis
- **savename** (Optional[str]) – Optional name of file to which to save the plot of feature importances
- **settings** (*PlotSettings*) – *PlotSettings* class to control figure appearance

Return type None

8.4 lumin.plotting.plot_settings module

class lumin.plotting.plot_settings.**PlotSettings**(***kargs*)
Bases: object

Class to provide control over plot appearances. Default parameters are set automatically, and can be adjusted by passing values as keyword arguments during initialisation (or changed after instantiation)

Parameters **arguments** (*keyword*) – used to set relevant plotting parameters

str2sz(*sz*, *ax*)

Used to map requested plot sizes to actual dimensions

Parameters

- **sz** (str) – string representation of size
- **ax** (str) – axis dimension requested

Return type float

Returns width of plot dimension

8.5 lumin.plotting.results module

lumin.plotting.results.**plot_roc**(*data*, *pred_name='pred'*, *targ_name='gen_target'*, *wgt_name=None*, *labels=None*, *plot_params=None*, *n_bootstrap=0*, *log_x=False*, *plot_baseline=True*, *savename=None*, *settings=<lumin.plotting.plot_settings.PlotSettings object>*)

Plot receiver operating characteristic curve(s), optionally using booststrap resampling

Parameters

- **data** (Union[DataFrame, List[DataFrame]]) – (list of) DataFrame(s) from which to draw predictions and targets
- **pred_name** (str) – name of column to use as predictions
- **targ_name** (str) – name of column to use as targets
- **wgt_name** (Optional[str]) – optional name of column to use as sample weights
- **labels** (Union[str, List[str], None]) – (list of) label(s) for plot legend
- **plot_params** (Union[Dict[str, Any], List[Dict[str, Any]], None]) – (list of) dictionar[y/ies] of argument(s) to pass to line plot
- **n_bootstrap** (int) – if greater than 0, will bootstrap resample the data that many times when computing the ROC AUC. Currently, this does not affect the shape of the lines, which are based on computing the ROC for the entire dataset as is.
- **log_x** (bool) – whether to use a log scale for plotting the x-axis, useful for high AUC line
- **plot_baseline** (bool) – whether to plot a dotted line for AUC=0.5. Currently incompatable with log_x=True
- **savename** (Optional[str]) – Optional name of file to which to save the plot of feature importances
- **settings** (*PlotSettings*) – *PlotSettings* class to control figure appearance

Return type Dict[str, Union[float, Tuple[float, float]]]

Returns Dictionary mapping data labels to aucs (and uncertainties if n_bootstrap > 0)

lumin.plotting.results.**plot_binary_class_pred**(*df*, *pred_name='pred'*, *targ_name='gen_target'*, *wgt_name=None*, *wgt_scale=1*, *log_y=False*, *lim_x=(0, 1)*, *density=True*, *savename=None*, *settings=<lumin.plotting.plot_settings.PlotSettings object>*)

Basic plotter for prediction distribution in a binary classification problem. Note that labels are set using the settings.targ2class dictionary, which by default is {0: 'Background', 1: 'Signal'}.

Parameters

- **df** (DataFrame) – DataFrame with targets and predictions
- **pred_name** (str) – name of column to use as predictions
- **targ_name** (str) – name of column to use as targets
- **wgt_name** (Optional[str]) – optional name of column to use as sample weights
- **wgt_scale** (float) – applies a global multiplicative rescaling to sample weights. Default 1 = no rescaling
- **log_y** (bool) – whether to use a log scale for the y-axis
- **lim_x** (Tuple[float, float]) – limit for plotting on the x-axis
- **density** – whether to normalise each distribution to one, or keep set to sum of weights / datapoints
- **savename** (Optional[str]) – Optional name of file to which to save the plot of feature importances
- **settings** (*PlotSettings*) – *PlotSettings* class to control figure appearance

Return type None

lumin.plotting.results.**plot_sample_pred**(*df*, *pred_name='pred'*, *targ_name='gen_target'*, *wgt_name='gen_weight'*, *sample_name='gen_sample'*, *wgt_scale=1*, *bins=35*, *log_y=True*, *lim_x=(0, 1)*, *density=False*, *zoom_args=None*, *savename=None*, *settings=<lumin.plotting.plot_settings.PlotSettings object>*)

More advanced plotter for prediction distribution in a binary class problem with stacked distributions for backgrounds and user-defined binning Can also zoom in to specified parts of plot Note that plotting colours can be controled by seeting the settings.sample2col dictionary

Parameters

- **df** (DataFrame) – DataFrame with targets and predictions
- **pred_name** (str) – name of column to use as predictions
- **targ_name** (str) – name of column to use as targets
- **wgt_name** (str) – name of column to use as sample weights
- **sample_name** (str) – name of column to use as process names
- **wgt_scale** (float) – applies a global multiplicative rescaling to sample weights. Default 1 = no rescaling

- **bins** (Union[int, List[int]]) – either the number of bins to use for a uniform binning, or a list of bin edges for a variable-width binning
- **log_y** (bool) – whether to use a log scale for the y-axis
- **lim_x** (Tuple[float, float]) – limit for plotting on the x-axis
- **density** – whether to normalise each distribution to one, or keep set to sum of weights / datapoints
- **zoom_args** (Optional[Dict[str, Any]]) – arguments to control the optional zoomed in section, e.g. {'x':(0.4,0.45), 'y':(0.2, 1500), 'anchor':(0,0.25,0.95,1), 'width_scale':1, 'width_zoom':4, 'height_zoom':3}
- **savename** (Optional[str]) – Optional name of file to which to save the plot of feature importances
- **settings** (*PlotSettings*) – *PlotSettings* class to control figure appearance

Return type None

8.6 lumin.plotting.training module

lumin.plotting.training.**plot_train_history**(*histories*, *savename=None*, *ignore_trn=False*, *settings=<lumin.plotting.plot_settings.PlotSettings object>*, *show=True*, *xlow=0*, *log_y=False*)

Plot histories object returned by *train_models()* showing the loss evolution over time per model trained.

Parameters

- **histories** (List[OrderedDict]) – list of dictionaries mapping loss type to values at each (sub)-epoch
- **savename** (Optional[str]) – Optional name of file to which to save the plot of feature importances
- **ignore_trn** (bool) – whether to ignore training loss
- **settings** (*PlotSettings*) – *PlotSettings* class to control figure appearance
- **show** (bool) – whether or not to show the plot, or just save it
- **xlow** (int) – if set, will cut out the first given number of epochs
- **log_y** (bool) – whether to plot the y-axis with a log scale

Return type None

lumin.plotting.training.**plot_lr_finders**(*lr_finders*, *lr_range=None*, *loss_range='auto'*, *log_y='auto'*, *savename=None*, *settings=<lumin.plotting.plot_settings.PlotSettings object>*)

Plot mean loss evolution against learning rate for several fold_lr_find.

Parameters

- **lr_finders** (List[AbsCallback]) – list of fold_lr_find)
- **lr_range** (Union[float, Tuple, None]) – limits the range of learning rates plotted on the x-axis: if float, maximum LR; if tuple, minimum & maximum LR

- **loss_range** (Union[float, Tuple, str, None]) – limits the range of losses plotted on the x-axis: if float, maximum loss; if tuple, minimum & maximum loss; if None, no limits; if 'auto', computes an upper limit automatically
- **log_y** (Union[str, bool]) – whether to plot y-axis as log. If 'auto', will set to log if maximal fractional difference in loss values is greater than 50
- **savename** (Optional[str]) – Optional name of file to which to save the plot
- **settings** (*PlotSettings*) – *PlotSettings* class to control figure appearance

Return type None

8.7 Module contents

CHAPTER

NINE

LUMIN.UTILS PACKAGE

9.1 Submodules

9.2 lumin.utils.data module

lumin.utils.data.**check_val_set**(*train*, *val*, *test=None*, *n_folds=None*)

Method to check validation set suitability by seeing whether Random Forests can predict whether events belong to one dataset or another. If a *FoldYielder* is passed, then trainings are run once per fold and averaged. Will compute the ROC AUC for set discrimination (should be close to 0.5) and compute the feature importances to aid removal of discriminating features.

Parameters

- **train** (Union[DataFrame, ndarray, *FoldYielder*]) – training data
- **val** (Union[DataFrame, ndarray, *FoldYielder*]) – validation data
- **test** (Union[DataFrame, ndarray, *FoldYielder*, None]) – optional testing data
- **n_folds** (Optional[int]) – if set and if passed a *FoldYielder*, will only use the first n_folds folds

Return type None

9.3 lumin.utils.misc module

lumin.utils.misc.**to_np**(*x*)

Convert Tensor x to a Numpy array

Parameters **x** (Tensor) – Tensor to convert

Return type ndarray

Returns x as a Numpy array

lumin.utils.misc.**to_device**(*x*, *device=device(type='cpu')*)

Recursively place Tensor(s) onto device

Parameters **x** (Union[Tensor, List[Tensor]]) – Tensor(s) to place on device

Return type Union[Tensor, List[Tensor]]

Returns Tensor(s) on device

lumin.utils.misc.**to_tensor**(*x*)

Convert Numpy array to Tensor with possibility of a None being passed

Parameters **x** (Optional[ndarray]) – Numpy array or None

Return type Optional[Tensor]

Returns x as Tensor or None

lumin.utils.misc.**str2bool**(*string*)

Convert string representation of Boolean to bool

Parameters **string** (Union[str, bool]) – string representation of Boolean (or a Boolean)

Return type bool

Returns bool if bool was passed else, True if lowercase string matches is in ("yes", "true", "t", "1")

lumin.utils.misc.**to_binary_class**(*df*, *zero_preds*, *one_preds*)

Map class precitions back to a binary prediction. The maximum prediction for features listed in zero_preds is treated as the prediction for class 0, vice versa for one_preds. The binary prediction is added to df in place as column 'pred'

Parameters

- **df** (DataFrame) – DataFrame containing prediction features
- **zero_preds** (List[str]) – list of column names for predictions associated with class 0
- **one_preds** (List[str]) – list of column names for predictions associated with class 0

Return type None

lumin.utils.misc.**ids2unique**(*ids*)

Map a permutaion of integers to a unique number, or a 2D array of integers to unique numbers by row. Returned numbers are unique for a given permutation of integers. This is achieved by computing the product of primes raised to powers equal to the integers. Beacause of this, it can be easy to produce numbers which are too large to be stored if many (large) integers are passed.

Parameters **ids** (Union[List[int], ndarray]) – (array of) permutation(s) of integers to map

Return type ndarray

Returns (Array of) unique id(s) for given permutation(s)

class lumin.utils.misc.**ForwardHook**(*module*, *hook_fn=None*)

Bases: object

Create a hook for performing an action based on the forward pass thorugh a nn.Module

Parameters

- **module** (Module) – nn.Module to hook
- **hook_fn** (Optional[Callable[[Module, Union[Tensor, Tuple[Tensor]], Union[Tensor, Tuple[Tensor]]], None]]) – Optional function to perform. Default is to record input and output of module

Examples::

```
>>> hook = ForwardHook(model.tail.dense)
>>> model.predict(inputs)
>>> print(hook.inputs)
```

hook_fn(*module*, *input*, *output*)

Default hook function records inputs and outputs of module

Parameters

- **module** (Module) – nn.Module to hook
- **input** (Union[Tensor, Tuple[Tensor]]) – input tensor
- **output** (Union[Tensor, Tuple[Tensor]]) – output tensor of module

Return type None

remove()
Call when finished to remove hook

Return type None

class lumin.utils.misc.**BackwardHook**(*module*, *hook_fn=None*)
Bases: *lumin.utils.misc.ForwardHook*

Create a hook for performing an action based on the backward pass thorugh a nn.Module

Parameters

- **module** (Module) – nn.Module to hook
- **hook_fn** (Optional[Callable[[Module, Union[Tensor, Tuple[Tensor]], Union[Tensor, Tuple[Tensor]]], None]]) – Optional function to perform. Default is to record input and output of module

Examples::

```
>>> hook = BackwardHook(model.tail.dense)
>>> model.predict(inputs)
>>> print(hook.inputs)
```

lumin.utils.misc.**subsample_df**(*df*, *objective*, *targ_name*, *n_samples=None*, *replace=False*, *strat_key=None*, *wgt_name=None*)
Subsamples, or samples with replacement, a DataFrame. Will automatically reweight data such that weight sums remain the same as the original DataFrame (per class)

Parameters

- **df** (DataFrame) – DataFrame to sample
- **objective** (str) – string representation of objective: either 'classification' or 'regression'
- **targ_name** (str) – name of column containing target data
- **n_samples** (Optional[int]) – If set, will sample that number of data points, otherwise will sample with replacement a new DataFRame of the same size as the original
- **replace** (bool) – whether to sample with replacement
- **strat_key** (Optional[str]) – column name to use for stratified subsampling, if desired
- **wgt_name** (Optional[str]) – name of column containing weight data. If set, will reweight subsampled data, otherwise will not

Return type DataFrame

lumin.utils.misc.**is_partially**(*var*)
Retuns true if var is partial, function, or class, else false.

Parameters **var** (Any) – variable to inspect

Return type bool

Returns true if var is partial or partialler, else false

9.4 lumin.utils.multiprocessing module

lumin.utils.multiprocessing.**mp_run**(*args*, *func*)

Run multiple instances of function simultaneously by using a list of argument dictionaries Runs given function once per entry in args list.

Important: Function should put a dictionary of results into the *mp.Queue* and each result key should be unique otherwise they will overwrite one another.

Parameters

- **args** (List[Dict[Any, Any]]) – list of dictionaries of arguments
- **func** (Callable[[Any], Any]) – function to which to pass dictionary arguments

Return type Dict[Any, Any]

Returns Dictionary of results

9.5 lumin.utils.statistics module

lumin.utils.statistics.**bootstrap_stats**(*args*, *out_q=None*)

Computes statistics and KDEs of data via sampling with replacement

Parameters

- **args** (Dict[str, Any]) – dictionary of arguments. Possible keys are: data - data to resample name - name prepended to returned keys in result dict weights - array of weights matching length of data to use for weighted resampling n - number of times to resample data x - points at which to compute the kde values of resample data kde - whether to compute the kde values at x-points for resampled data mean - whether to compute the means of the resampled data std - whether to compute standard deviation of resampled data c68 - whether to compute the width of the absolute central 68.2 percentile of the resampled data
- **out_q** (Optional[<bound method BaseContext.Queue of <multiprocessing.context.DefaultContext object at 0x7f97a0c10748>>]) – if using multiporcessing can place result dictionary in provided queue

Return type Union[None, Dict[str, Any]]

Returns Result dictionary if *out_q* is *None* else *None*.

lumin.utils.statistics.**get_moments**(*arr*)

Computes mean and std of data, and their associated uncertainties

Parameters **arr** (ndarray) – univariate data

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

Returns

- mean
- statistical uncertainty of mean
- standard deviation
- statistical uncertainty of standard deviation

lumin.utils.statistics.**uncert_round**(*value*, *uncert*)

Round value according to given uncertainty using one significant figure of the uncertainty

Parameters

- **value** (float) – value to round
- **uncert** (float) – uncertainty of value

Return type Tuple[float, float]

Returns

- rounded value
- rounded uncertainty

9.6 Module contents

CHAPTER

TEN

PACKAGE DESCRIPTION

10.1 Distinguishing Characteristics

10.1.1 Data objects

- Use with large datasets: HEP data can become quite large, making training difficult:
  - The `FoldYielder` class provides on-demand access to data stored in HDF5 format, only loading into memory what is required.
  - Conversion from ROOT and CSV to HDF5 is easy to achieve using (see examples)
  - `FoldYielder` provides conversion methods to Pandas `DataFrame` for use with other internal methods and external packages
- Non-network-specific methods expect Pandas `DataFrame` allowing their use without having to convert to `FoldYielder`.

10.1.2 Deep learning

- PyTorch > 1.0
- Inclusion of recent deep learning techniques and practices, including:
  - Dynamic learning rate, momentum, beta_1:
    - Cyclical, Smith, 2015
    - Cosine annealed Loschilov & Hutter, 2016
    - 1-cycle, Smith, 2018
  - HEP-specific data augmentation during training and inference
  - Advanced ensembling methods:
    - Snapshot ensembles Huang et al., 2017
    - Fast geometric ensembles Garipov et al., 2018
    - Stochastic Weight Averaging Izmailov et al., 2018
  - Learning Rate Finders, Smith, 2015
  - Entity embedding of categorical features, Guo & Berkhahn, 2016
  - Label smoothing Szegedy et al., 2015
  - Running batchnorm fastai 2019

- Flexible architecture construction:
  - `ModelBuilder` takes parameters and modules to yield networks on-demand
  - Networks constructed from modular blocks:
    - Head - Takes input features
    - Body - Contains most of the hidden layers
    - Tail - Scales down the body to the desired number of outputs
    - Endcap - Optional layer for use post-training to provide further computation on model outputs; useful when training on a proxy objective
  - Easy loading and saving of pre-trained embedding weights
  - Modern architectures like:
    - Residual and dense(-like) networks (He et al. 2015 & Huang et al. 2016)
    - Graph nets for physics objects, e.g. Battaglia, Pascanu, Lai, Rezende, Kavukcuoglu, 2016, Moreno et al., 2019, and Qasim, Kieseler, Iiyama, & Pierini, 2019, with optional self-attention Vaswani et al., 2017.
    - Recurrent layers for series of objects
    - 1D convolutional networks for series of objects
    - Squeeze-excitation blocks Hu, Shen, Albanie, Sun, & Wu, 2017
    - HEP-specific architectures, e.g. LorentzBoostNetworks Erdmann, Geiser, Rath, Rieger, 2018
- Configurable initialisations, including LSUV Mishkin, Matas, 2016
- HEP-specific losses, e.g. Asimov loss Elwood & Krücker, 2018
- Exotic training schemes, e.g. Learning to Pivot with Adversarial Networks Louppe, Kagan, & Cranmer, 2016
- Easy training and inference of ensembles of models:
  - Default training method `fold_train_ensemble`, trains a specified number of models as well as just a single model
  - `Ensemble` class handles the (metric-weighted) construction of an ensemble, its inference, saving and loading, and interpretation
- Easy exporting of models to other libraries via Onnx
- Use with CPU and NVidia GPU
- Evaluation on domain-specific metrics such as Approximate Median Significance via `EvalMetric` class
- fastai-style callbacks and stateful model-fitting, allowing training, models, losses, and data to be accessible and adjustable at any point

10.1.3 Feature selection methods

- Dendrograms of feature-pair monotonicity
- Feature importance via auto-optimised SK-Learn random forests
- Mutual dependence (via RFPImp)
- Automatic filtering and selection of features

10.1.4 Interpretation

- Feature importance for models and ensembles
- Embedding visualisation
- 1D & 2D partial dependency plots (via PDPbox)

10.1.5 Plotting

- Variety of domain-specific plotting functions
- Unified appearance via `PlotSettings` class - class accepted by every plot function providing control of plot appearance, titles, colour schemes, et cetera

10.1.6 Universal handling of sample weights

- HEP events are normally accompanied by weight characterising the acceptance and production cross-section of that particular event, or to flatten some distribution.
- Relevant methods and classes can take account of these weights.
- This includes training, interpretation, and plotting
- Expansion of PyTorch losses to better handle weights

10.1.7 Parameter optimisation

- Optimal learning rate via cross-validated range tests Smith, 2015
- Quick, rough optimisation of random forest hyper parameters
- Generalisable Cut & Count thresholds
- 1D discriminant binning with respect to bin-fill uncertainty

10.1.8 Statistics and uncertainties

- Integral to experimental science
- Quantitative results are accompanied by uncertainties
- Use of bootstrapping to improve precision of statistics estimated from small samples

10.1.9 Look and feel

- LUMIN aims to feel fast to use - liberal use of progress bars mean you're able to always know when tasks will finish, and get live updates of training
- Guaranteed to spark joy (in its current beta state, LUMIN may instead ignite rage, despair, and frustration - *dev.*)

10.2 Notes

10.2.1 Why use LUMIN

TMVA contained in CERN's ROOT system, has been the default choice for BDT training for analysis and reconstruction algorithms due to never having to leave ROOT format. With the gradual move to DNN approaches, more scientists are looking to move their data out of ROOT to use the wider selection of tools which are available. Keras appears to be the first stop due to its ease of use, however implementing recent methods in Keras can be difficult, and sometimes requires dropping back to the tensor library that it aims to abstract. Indeed, the prequel to LUMIN was a similar wrapper for Keras (HEPML_Tools) which involved some pretty ugly hacks. The fastai framework provides access to these recent methods, however doesn't yet support sample weights to the extent that HEP requires. LUMIN aims to provide the best of both, Keras-style sample weighting and fastai training methods, while focussing on columnar data and providing domain-specific metrics, plotting, and statistical treatment of results and uncertainties.

10.2.2 Data types

LUMIN is primarily designed for use on columnar data, and from version 0.5 onwards this also includes *matrix data*; ordered series and un-ordered groups of objects. With some extra work it can be used on other data formats, but at the moment it has nothing special to offer. Whilst recent work in HEP has made use of jet images and GANs, these normally hijack existing ideas and models. Perhaps once we get established, domain specific approaches which necessitate the use of a specialised framework, then LUMIN could grow to meet those demands, but for now I'd recommend checking out the fastai library, especially for image data.

With just one main developer, I'm simply focussing on the data types and applications I need for my own research and common use cases in HEP. If, however you would like to use LUMIN's other methods for your own work on other data formats, then you are most welcome to contribute and help to grow LUMIN to better meet the needs of the scientific community.

10.2.3 Future

The current priority is to improve the documentation, add unit tests, and expand the examples.

The next step will be to try to increase the user base and number of contributors. I'm aiming to achieve this through presentations, tutorials, blog posts, and papers.

Further improvements will be in the direction of implementing new methods and (HEP-specific) architectures, as well as providing helper functions and data exporters to statistical analysis packages like Combine and PYHF.

10.2.4 Contributing & feedback

Contributions, suggestions, and feedback are most welcome! The issue tracker on this repo is probably the best place to report bugs et cetera.

10.2.5 Code style

Nope, the majority of the code-base does not conform to PEP8. PEP8 has its uses, but my understanding is that it primarily written for developers and maintainers of software whose users never need to read the source code. As a maths-heavy research framework which users are expected to interact with, PEP8 isn't the best style. Instead, I'm aiming to follow more the style of fastai, which emphasises, in particular, reducing vertical space (useful for reading source code in a notebook) naming and abbreviating variables according to their importance and lifetime (easier to

recognise which variables have a larger scope and permits easier writing of mathematical operations). A full list of the abbreviations used may be found in abbr.md

10.2.6 Why is LUMIN called LUMIN?

Aside from being a recursive acronym (and therefore the best kind of acronym) lumin is short for 'luminosity'. In high-energy physics, the integrated luminosity of the data collected by an experiment is the main driver in the results that analyses obtain. With the paradigm shift towards multivariate analyses, however, improved methods can be seen as providing 'artificial luminosity'; e.g. the gain offered by some DNN could be measured in terms of the amount of extra data that would have to be collected to achieve the same result with a more traditional analysis. Luminosity can also be connected to the fact that LUMIN is built around the python version of Torch.

10.2.7 Who develops LUMIN

LUMIN is primarily developed by Giles Strong; a British-born doctor in particle physics, researcher at The University of Padova (Italy), and a member of the CMS collaboration at CERN.

As LUMIN has grown, it has welcomed contributions from members of the scientific and software development community. Check out the contributors page for a complete list.

Certainly more developers and contributors are welcome to join and help out!

10.2.8 Reference

If you have used LUMIN in your analysis work and wish to cite it, the preferred reference is: *Giles C. Strong, LUMIN, Zenodo (Mar. 2019), https://doi.org/10.5281/zenodo.2601857, Note: Please check https://github.com/GilesStrong/lumin/graphs/contributors for the full list of contributors*

```
@misc{giles_chatham_strong_2019_2601857,
  author       = {Giles Chatham Strong},
  title        = {LUMIN},
  month        = mar,
  year         = 2019,
  note         = {{Please check https://github.com/GilesStrong/lumin/graphs/
↪contributors for the full list of contributors}},
  doi          = {10.5281/zenodo.2601857},
  url          = {https://doi.org/10.5281/zenodo.2601857}
}
```

CHAPTER

ELEVEN

INDEX

- genindex

PYTHON MODULE INDEX

l

lumin.data_processing, 21
lumin.data_processing.file_proc, 11
lumin.data_processing.hep_proc, 14
lumin.data_processing.pre_proc, 19
lumin.evaluation, 24
lumin.evaluation.ams, 23
lumin.inference, 26
lumin.inference.summary_stat, 25
lumin.nn, 110
lumin.nn.callbacks, 40
lumin.nn.callbacks.adversarial_callbacks, 27
lumin.nn.callbacks.callback, 28
lumin.nn.callbacks.cyclic_callbacks, 29
lumin.nn.callbacks.data_callbacks, 32
lumin.nn.callbacks.loss_callbacks, 34
lumin.nn.callbacks.lsuv_init, 34
lumin.nn.callbacks.model_callbacks, 35
lumin.nn.callbacks.monitors, 37
lumin.nn.callbacks.opt_callbacks, 39
lumin.nn.callbacks.pred_handlers, 40
lumin.nn.data, 46
lumin.nn.data.batch_yielder, 40
lumin.nn.data.fold_yielder, 41
lumin.nn.ensemble, 51
lumin.nn.ensemble.ensemble, 46
lumin.nn.interpretation, 52
lumin.nn.interpretation.features, 51
lumin.nn.losses, 56
lumin.nn.losses.advanced_losses, 52
lumin.nn.losses.basic_weighted, 54
lumin.nn.losses.hep_losses, 56
lumin.nn.metrics, 63
lumin.nn.metrics.class_eval, 56
lumin.nn.metrics.eval_metric, 60
lumin.nn.metrics.reg_eval, 61
lumin.nn.models, 108
lumin.nn.models.blocks, 94
lumin.nn.models.blocks.body, 63
lumin.nn.models.blocks.conv_blocks, 67
lumin.nn.models.blocks.endcap, 73
lumin.nn.models.blocks.gnn_blocks, 74
lumin.nn.models.blocks.head, 80
lumin.nn.models.blocks.tail, 93
lumin.nn.models.helpers, 98
lumin.nn.models.initialisations, 99
lumin.nn.models.layers, 98
lumin.nn.models.layers.activations, 95
lumin.nn.models.layers.batchnorms, 95
lumin.nn.models.layers.mish, 97
lumin.nn.models.layers.self_attention, 97
lumin.nn.models.model, 100
lumin.nn.models.model_builder, 104
lumin.nn.training, 110
lumin.nn.training.train, 108
lumin.optimisation, 120
lumin.optimisation.features, 111
lumin.optimisation.hyper_param, 118
lumin.optimisation.threshold, 119
lumin.plotting, 131
lumin.plotting.data_viewing, 121
lumin.plotting.interpretation, 124
lumin.plotting.plot_settings, 128
lumin.plotting.results, 128
lumin.plotting.training, 130
lumin.utils, 137
lumin.utils.data, 133
lumin.utils.misc, 133
lumin.utils.multiprocessing, 136
lumin.utils.statistics, 136

INDEX

A

AbsConv1dHead (*class in lumin.nn.models.blocks.head*), 85
AbsCyclicCallback (*class in lumin.nn.callbacks.cyclic_callbacks*), 29
AbsEndcap (*class in lumin.nn.models.blocks.endcap*), 73
AdaptiveAvgMaxConcatPool1d (*class in lumin.nn.models.blocks.conv_blocks*), 71
AdaptiveAvgMaxConcatPool2d (*class in lumin.nn.models.blocks.conv_blocks*), 71
AdaptiveAvgMaxConcatPool3d (*class in lumin.nn.models.blocks.conv_blocks*), 71
add_abs_mom() (*in module lumin.data_processing.hep_proc*), 15
add_energy() (*in module lumin.data_processing.hep_proc*), 15
add_ignore() (*lumin.nn.data.fold_yielder.FoldYielder method*), 41
add_input_pipe() (*lumin.nn.data.fold_yielder.FoldYielder method*), 41
add_input_pipe() (*lumin.nn.ensemble.ensemble.Ensemble method*), 47
add_input_pipe_from_file() (*lumin.nn.data.fold_yielder.FoldYielder method*), 42
add_mass() (*in module lumin.data_processing.hep_proc*), 15
add_matrix_pipe() (*lumin.nn.data.fold_yielder.FoldYielder method*), 42
add_matrix_pipe_from_file() (*lumin.nn.data.fold_yielder.FoldYielder method*), 42
add_meta_data() (*in module lumin.data_processing.file_proc*), 13
add_mt() (*in module lumin.data_processing.hep_proc*), 15
add_output_pipe() (*lumin.nn.data.fold_yielder.FoldYielder method*), 42
add_output_pipe() (*lumin.nn.ensemble.ensemble.Ensemble method*), 47
add_output_pipe_from_file() (*lumin.nn.data.fold_yielder.FoldYielder method*), 42
AMS (*class in lumin.nn.metrics.class_eval*), 56
ams_scan_quick() (*in module lumin.evaluation.ams*), 23
ams_scan_slow() (*in module lumin.evaluation.ams*), 24
auto_filter_on_linear_correlation() (*in module lumin.optimisation.features*), 114
auto_filter_on_mutual_dependence() (*in module lumin.optimisation.features*), 116
AutoExtractLorentzBoostNet (*class in lumin.nn.models.blocks.head*), 91

B

BackwardHook (*class in lumin.utils.misc*), 135
BatchYielder (*class in lumin.nn.data.batch_yielder*), 40
bin_binary_class_pred() (*in module lumin.inference.summary_stat*), 25
binary_class_cut_by_ams() (*in module lumin.optimisation.threshold*), 119
BinaryAccuracy (*class in lumin.nn.metrics.class_eval*), 58
BinaryLabelSmooth (*class in lumin.nn.callbacks.data_callbacks*), 32
boost() (*in module lumin.data_processing.hep_proc*), 17
boost2cm() (*in module lumin.data_processing.hep_proc*), 17
bootstrap_stats() (*in module lumin.utils.statistics*), 136
BootstrapResample (*class in lumin.nn.callbacks.data_callbacks*), 32
build_model() (*lumin.nn.models.model_builder.ModelBuilder method*), 106

C

calc_ams() (*in module lumin.evaluation.ams*), 23
calc_ams_torch() (*in module lumin.evaluation.ams*), 23
calc_emb_szs() (*lumin.nn.models.helpers.CatEmbedder method*), 99
calc_pair_mass() (*in module lumin.data_processing.hep_proc*), 17
Callback (*class in lumin.nn.callbacks.callback*), 28
CatEmbedder (*class in lumin.nn.models.helpers*), 98
CatEmbHead (*class in lumin.nn.models.blocks.head*), 80
check_out_sz() (*lumin.nn.models.blocks.head.AbsConv1dHead method*), 87
check_out_sz() (*lumin.nn.models.blocks.head.LorentzBoostNet method*), 90
check_val_set() (*in module lumin.utils.data*), 133
ClassRegMulti (*class in lumin.nn.models.blocks.tail*), 93
close() (*lumin.nn.data.fold_yielder.FoldYielder method*), 42
columns() (*lumin.nn.data.fold_yielder.FoldYielder method*), 42
compare_events() (*in module lumin.plotting.data_viewing*), 122
Conv1DBlock (*class in lumin.nn.models.blocks.conv_blocks*), 67
cos_delta() (*in module lumin.data_processing.hep_proc*), 18
CycleLR (*class in lumin.nn.callbacks.cyclic_callbacks*), 30
CycleMom (*class in lumin.nn.callbacks.cyclic_callbacks*), 30
CycleStep (*class in lumin.nn.callbacks.cyclic_callbacks*), 31

D

delta_phi() (*in module lumin.data_processing.hep_proc*), 14
delta_r() (*in module lumin.data_processing.hep_proc*), 18
delta_r_boosted() (*in module lumin.data_processing.hep_proc*), 18
df2foldfile() (*in module lumin.data_processing.file_proc*), 12

E

EarlyStopping (*class in lumin.nn.callbacks.monitors*), 37
Ensemble (*class in lumin.nn.ensemble.ensemble*), 46
EpochSaver (*class in lumin.nn.callbacks.monitors*), 39
EvalMetric (*class in lumin.nn.metrics.eval_metric*), 60
evaluate() (*lumin.nn.metrics.class_eval.AMS method*), 57
evaluate() (*lumin.nn.metrics.class_eval.BinaryAccuracy method*), 58
evaluate() (*lumin.nn.metrics.class_eval.MultiAMS method*), 58
evaluate() (*lumin.nn.metrics.class_eval.RocAucScore method*), 59
evaluate() (*lumin.nn.metrics.eval_metric.EvalMetric method*), 60
evaluate() (*lumin.nn.metrics.reg_eval.RegAsProxyPull method*), 63
evaluate() (*lumin.nn.metrics.reg_eval.RegPull method*), 62
evaluate() (*lumin.nn.models.model.Model method*), 100
evaluate_model() (*lumin.nn.metrics.eval_metric.EvalMetric method*), 60
evaluate_preds() (*lumin.nn.metrics.eval_metric.EvalMetric method*), 60
event_to_cartesian() (*in module lumin.data_processing.hep_proc*), 16
export2onnx() (*lumin.nn.ensemble.ensemble.Ensemble method*), 47
export2onnx() (*lumin.nn.models.model.Model method*), 100
export2tfpb() (*lumin.nn.ensemble.ensemble.Ensemble method*), 47
export2tfpb() (*lumin.nn.models.model.Model method*), 101

F

feat_extractor() (*lumin.nn.models.blocks.head.AutoExtractLorentzBoostNet method*), 92
feat_extractor() (*lumin.nn.models.blocks.head.LorentzBoostNet method*), 90
fit() (*lumin.nn.models.model.Model method*), 101
fit_input_pipe() (*in module lumin.data_processing.pre_proc*), 19
fit_output_pipe() (*in module lumin.data_processing.pre_proc*), 20
fix_event_phi() (*in module lumin.data_processing.hep_proc*), 15
fix_event_y() (*in module lumin.data_processing.hep_proc*), 16

fix_event_z() (*in module lumin.data_processing.hep_proc*), 16
fold2foldfile() (*in module lumin.data_processing.file_proc*), 11
FoldYielder (*class in lumin.nn.data.fold_yielder*), 41
forward() (*lumin.nn.losses.advanced_losses.WeightedBinnedHuber method*), 53
forward() (*lumin.nn.losses.advanced_losses.WeightedFractionalBinnedHuber method*), 54
forward() (*lumin.nn.losses.advanced_losses.WeightedFractionalMSE method*), 53
forward() (*lumin.nn.losses.basic_weighted.WeightedCCE method*), 55
forward() (*lumin.nn.losses.basic_weighted.WeightedMAE method*), 55
forward() (*lumin.nn.losses.basic_weighted.WeightedMSE method*), 54
forward() (*lumin.nn.losses.hep_losses.SignificanceLoss method*), 56
forward() (*lumin.nn.models.blocks.body.FullyConnected method*), 65
forward() (*lumin.nn.models.blocks.body.IdentBody method*), 63
forward() (*lumin.nn.models.blocks.body.MultiBlock method*), 66
forward() (*lumin.nn.models.blocks.conv_blocks.AdaptiveAvgMaxConcatPool method*), 71
forward() (*lumin.nn.models.blocks.conv_blocks.Conv1DBlock method*), 67
forward() (*lumin.nn.models.blocks.conv_blocks.Res1DBlock method*), 69
forward() (*lumin.nn.models.blocks.conv_blocks.ResNeXt1DBlock method*), 70
forward() (*lumin.nn.models.blocks.conv_blocks.SEBlock1d method*), 72
forward() (*lumin.nn.models.blocks.endcap.AbsEndcap method*), 73
forward() (*lumin.nn.models.blocks.gnn_blocks.GraphCoAttnBlock method*), 75
forward() (*lumin.nn.models.blocks.gnn_blocks.GravNet method*), 79
forward() (*lumin.nn.models.blocks.gnn_blocks.GravNetLayer method*), 80
forward() (*lumin.nn.models.blocks.gnn_blocks.InteractionNet method*), 77
forward() (*lumin.nn.models.blocks.head.AbsConv1dHead method*), 87
forward() (*lumin.nn.models.blocks.head.CatEmbHead method*), 81
forward() (*lumin.nn.models.blocks.head.GNNHead method*), 83
forward() (*lumin.nn.models.blocks.head.LorentzBoostNet method*), 91
forward() (*lumin.nn.models.blocks.head.MultiHead method*), 82
forward() (*lumin.nn.models.blocks.head.RecurrentHead method*), 85
forward() (*lumin.nn.models.blocks.tail.ClassRegMulti method*), 94
forward() (*lumin.nn.models.blocks.tail.IdentTail method*), 93
forward() (*lumin.nn.models.layers.activations.Swish method*), 95
forward() (*lumin.nn.models.layers.batchnorms.LCBatchNorm1d method*), 95
forward() (*lumin.nn.models.layers.batchnorms.RunningBatchNorm1d method*), 96
forward() (*lumin.nn.models.layers.batchnorms.RunningBatchNorm2d method*), 96
forward() (*lumin.nn.models.layers.mish.Mish method*), 97
forward() (*lumin.nn.models.layers.self_attention.SelfAttention method*), 98
ForwardHook (*class in lumin.utils.misc*), 134
freeze_layers() (*lumin.nn.models.model.Model method*), 101
from_fy() (*lumin.nn.models.helpers.CatEmbedder class method*), 99
from_model_builder() (*lumin.nn.models.model_builder.ModelBuilder class method*), 106
from_models() (*lumin.nn.ensemble.ensemble.Ensemble class method*), 47
from_results() (*lumin.nn.ensemble.ensemble.Ensemble class method*), 48
from_save() (*lumin.nn.ensemble.ensemble.Ensemble class method*), 49
from_save() (*lumin.nn.models.model.Model class method*), 101
FullyConnected (*class in lumin.nn.models.blocks.body*), 64
func() (*lumin.nn.models.blocks.endcap.AbsEndcap method*), 73

G

get_body() (*lumin.nn.models.model_builder.ModelBuilder method*), 107
get_column() (*lumin.nn.data.fold_yielder.FoldYielder method*), 42
get_conv1d_block() (*lumin.nn.models.blocks.head.AbsConv1dHead method*), 87
get_conv1d_res_block() (*lumin.nn.models.blocks.head.AbsConv1dHead method*), 88

get_conv1d_resNeXt_block() (*lumin.nn.models.blocks.head.AbsConv1dHead method*), 88
get_conv_layer() (*lumin.nn.models.blocks.conv_blocks.Conv1DBlock method*), 67
get_data() (*lumin.nn.data.fold_yielder.FoldYielder method*), 43
get_data_count() (*lumin.nn.data.fold_yielder.FoldYielder method*), 43
get_df() (*lumin.nn.callbacks.opt_callbacks.LRFinder method*), 39
get_df() (*lumin.nn.data.fold_yielder.FoldYielder method*), 43
get_df() (*lumin.nn.metrics.eval_metric.EvalMetric method*), 61
get_embeds() (*lumin.nn.models.blocks.head.CatEmbHead method*), 81
get_ensemble_feat_importance() (*in module lumin.nn.interpretation.features*), 51
get_feat_importance() (*lumin.nn.ensemble.ensemble.Ensemble method*), 49
get_feat_importance() (*lumin.nn.models.model.Model method*), 102
get_fold() (*lumin.nn.data.fold_yielder.FoldYielder method*), 44
get_fold() (*lumin.nn.data.fold_yielder.HEPAugFoldYielder method*), 46
get_head() (*lumin.nn.models.model_builder.ModelBuilder method*), 107
get_ignore() (*lumin.nn.data.fold_yielder.FoldYielder method*), 44
get_inputs() (*lumin.nn.data.batch_yielder.BatchYielder method*), 40
get_layers() (*lumin.nn.models.blocks.head.AbsConv1dHead method*), 89
get_loss() (*lumin.nn.callbacks.model_callbacks.SWA method*), 36
get_loss_history() (*lumin.nn.callbacks.monitors.MetricLogger method*), 38
get_lr() (*lumin.nn.models.model.Model method*), 102
get_metric() (*lumin.nn.metrics.eval_metric.EvalMetric method*), 61
get_model() (*lumin.nn.models.model_builder.ModelBuilder method*), 107
get_mom() (*lumin.nn.models.model.Model method*), 102
get_moments() (*in module lumin.utils.statistics*), 136
get_momentum() (*in module lumin.data_processing.hep_proc*), 18
get_nn_feat_importance() (*in module lumin.nn.interpretation.features*), 51
get_opt_rf_params() (*in module lumin.optimisation.hyper_param*), 118
get_out_size() (*lumin.nn.models.blocks.body.FullyConnected method*), 65
get_out_size() (*lumin.nn.models.blocks.body.IdentBody method*), 63
get_out_size() (*lumin.nn.models.blocks.body.MultiBlock method*), 66
get_out_size() (*lumin.nn.models.blocks.gnn_blocks.GraphCollapser method*), 75
get_out_size() (*lumin.nn.models.blocks.gnn_blocks.GravNet method*), 79
get_out_size() (*lumin.nn.models.blocks.gnn_blocks.GravNetLayer method*), 80
get_out_size() (*lumin.nn.models.blocks.gnn_blocks.InteractionNet method*), 77
get_out_size() (*lumin.nn.models.blocks.head.AbsConv1dHead method*), 89
get_out_size() (*lumin.nn.models.blocks.head.CatEmbHead method*), 81
get_out_size() (*lumin.nn.models.blocks.head.GNNHead method*), 84
get_out_size() (*lumin.nn.models.blocks.head.LorentzBoostNet method*), 91
get_out_size() (*lumin.nn.models.blocks.head.MultiHead method*), 83
get_out_size() (*lumin.nn.models.blocks.head.RecurrentHead method*), 85
get_out_size() (*lumin.nn.models.blocks.tail.ClassRegMulti method*), 94
get_out_size() (*lumin.nn.models.blocks.tail.IdentTail method*), 93
get_out_size() (*lumin.nn.models.layers.self_attention.SelfAttention method*), 98
get_out_size() (*lumin.nn.models.model.Model method*), 102
get_out_size() (*lu-*

min.nn.models.model_builder.ModelBuilder method), 107
get_padding() (*lumin.nn.models.blocks.conv_blocks.Conv1DBlock static method*), 68
get_param_count() (*lumin.nn.models.model.Model method*), 102
get_pre_proc_pipes() (*in module lumin.data_processing.pre_proc*), 19
get_preds() (*lumin.nn.callbacks.pred_handlers.PredHandler method*), 40
get_results() (*lumin.nn.callbacks.monitors.MetricLogger method*), 38
get_rf_feat_importance() (*in module lumin.optimisation.features*), 111
get_tail() (*lumin.nn.models.model_builder.ModelBuilder method*), 107
get_test_fold() (*lumin.nn.data.fold_yielder.HEPAugFoldYielder method*), 46
get_use_cat_feats() (*lumin.nn.data.fold_yielder.FoldYielder method*), 44
get_use_cont_feats() (*lumin.nn.data.fold_yielder.FoldYielder method*), 44
get_vecs() (*in module lumin.data_processing.hep_proc*), 15
get_weights() (*lumin.nn.models.model.Model method*), 102
GNNHead (*class in lumin.nn.models.blocks.head*), 83
GradClip (*class in lumin.nn.callbacks.loss_callbacks*), 34
GraphCollapser (*class in lumin.nn.models.blocks.gnn_blocks*), 74
GravNet (*class in lumin.nn.models.blocks.gnn_blocks*), 77
GravNetLayer (*class in lumin.nn.models.blocks.gnn_blocks*), 79

H

HEPAugFoldYielder (*class in lumin.nn.data.fold_yielder*), 44
hook_fn() (*lumin.utils.misc.ForwardHook method*), 134

I

IdentBody (*class in lumin.nn.models.blocks.body*), 63
IdentTail (*class in lumin.nn.models.blocks.tail*), 93
ids2unique() (*in module lumin.utils.misc*), 134
InteractionNet (*class in lumin.nn.models.blocks.gnn_blocks*), 76
is_partially() (*in module lumin.utils.misc*), 135

L

LCBatchNorm1d (*class in lumin.nn.models.layers.batchnorms*), 95
load() (*lumin.nn.ensemble.ensemble.Ensemble method*), 49
load() (*lumin.nn.models.model.Model method*), 103
load_pretrained() (*lumin.nn.models.model_builder.ModelBuilder method*), 107
load_trained_model() (*lumin.nn.ensemble.ensemble.Ensemble static method*), 50
lookup_act() (*in module lumin.nn.models.layers.activations*), 95
lookup_normal_init() (*in module lumin.nn.models.initialisations*), 99
lookup_uniform_init() (*in module lumin.nn.models.initialisations*), 100
LorentzBoostNet (*class in lumin.nn.models.blocks.head*), 89
lr_find() (*in module lumin.optimisation.hyper_param*), 118
LRFinder (*class in lumin.nn.callbacks.opt_callbacks*), 39
LsuvInit (*class in lumin.nn.callbacks.lsuv_init*), 35
lumin.data_processing (*module*), 21
lumin.data_processing.file_proc (*module*), 11
lumin.data_processing.hep_proc (*module*), 14
lumin.data_processing.pre_proc (*module*), 19
lumin.evaluation (*module*), 24
lumin.evaluation.ams (*module*), 23
lumin.inference (*module*), 26
lumin.inference.summary_stat (*module*), 25
lumin.nn (*module*), 110
lumin.nn.callbacks (*module*), 40
lumin.nn.callbacks.adversarial_callbacks (*module*), 27
lumin.nn.callbacks.callback (*module*), 28
lumin.nn.callbacks.cyclic_callbacks (*module*), 29
lumin.nn.callbacks.data_callbacks (*module*), 32
lumin.nn.callbacks.loss_callbacks (*module*), 34
lumin.nn.callbacks.lsuv_init (*module*), 34
lumin.nn.callbacks.model_callbacks (*module*), 35
lumin.nn.callbacks.monitors (*module*), 37
lumin.nn.callbacks.opt_callbacks (*module*), 39

lumin.nn.callbacks.pred_handlers (*module*), 40
lumin.nn.data (*module*), 46
lumin.nn.data.batch_yielder (*module*), 40
lumin.nn.data.fold_yielder (*module*), 41
lumin.nn.ensemble (*module*), 51
lumin.nn.ensemble.ensemble (*module*), 46
lumin.nn.interpretation (*module*), 52
lumin.nn.interpretation.features (*module*), 51
lumin.nn.losses (*module*), 56
lumin.nn.losses.advanced_losses (*module*), 52
lumin.nn.losses.basic_weighted (*module*), 54
lumin.nn.losses.hep_losses (*module*), 56
lumin.nn.metrics (*module*), 63
lumin.nn.metrics.class_eval (*module*), 56
lumin.nn.metrics.eval_metric (*module*), 60
lumin.nn.metrics.reg_eval (*module*), 61
lumin.nn.models (*module*), 108
lumin.nn.models.blocks (*module*), 94
lumin.nn.models.blocks.body (*module*), 63
lumin.nn.models.blocks.conv_blocks (*module*), 67
lumin.nn.models.blocks.endcap (*module*), 73
lumin.nn.models.blocks.gnn_blocks (*module*), 74
lumin.nn.models.blocks.head (*module*), 80
lumin.nn.models.blocks.tail (*module*), 93
lumin.nn.models.helpers (*module*), 98
lumin.nn.models.initialisations (*module*), 99
lumin.nn.models.layers (*module*), 98
lumin.nn.models.layers.activations (*module*), 95
lumin.nn.models.layers.batchnorms (*module*), 95
lumin.nn.models.layers.mish (*module*), 97
lumin.nn.models.layers.self_attention (*module*), 97
lumin.nn.models.model (*module*), 100
lumin.nn.models.model_builder (*module*), 104
lumin.nn.training (*module*), 110
lumin.nn.training.train (*module*), 108
lumin.optimisation (*module*), 120
lumin.optimisation.features (*module*), 111
lumin.optimisation.hyper_param (*module*), 118
lumin.optimisation.threshold (*module*), 119
lumin.plotting (*module*), 131
lumin.plotting.data_viewing (*module*), 121
lumin.plotting.interpretation (*module*), 124
lumin.plotting.plot_settings (*module*), 128
lumin.plotting.results (*module*), 128
lumin.plotting.training (*module*), 130
lumin.utils (*module*), 137
lumin.utils.data (*module*), 133
lumin.utils.misc (*module*), 133
lumin.utils.multiprocessing (*module*), 136
lumin.utils.statistics (*module*), 136

M

MetricLogger (*class in lumin.nn.callbacks.monitors*), 37
Mish (*class in lumin.nn.models.layers.mish*), 97
Model (*class in lumin.nn.models.model*), 100
ModelBuilder (*class in lumin.nn.models.model_builder*), 104
mp_run() (*in module lumin.utils.multiprocessing*), 136
MultiAMS (*class in lumin.nn.metrics.class_eval*), 57
MultiBlock (*class in lumin.nn.models.blocks.body*), 65
MultiHead (*class in lumin.nn.models.blocks.head*), 82

N

NodePredictor (*class in lumin.nn.models.blocks.gnn_blocks*), 75

O

on_backwards_end() (*lumin.nn.callbacks.loss_callbacks.GradClip method*), 34
on_batch_begin() (*lumin.nn.callbacks.adversarial_callbacks.PivotTraining method*), 28
on_batch_begin() (*lumin.nn.callbacks.cyclic_callbacks.AbsCyclicCallback method*), 29
on_batch_begin() (*lumin.nn.callbacks.cyclic_callbacks.CycleStep method*), 32
on_batch_begin() (*lumin.nn.callbacks.cyclic_callbacks.OneCycle method*), 31
on_batch_end() (*lumin.nn.callbacks.cyclic_callbacks.AbsCyclicCallback method*), 29
on_batch_end() (*lumin.nn.callbacks.monitors.MetricLogger method*), 38
on_batch_end() (*lumin.nn.callbacks.opt_callbacks.LRFinder method*), 39

on_epoch_begin() (lumin.nn.callbacks.adversarial_callbacks.PivotTraining method), 28
on_epoch_begin() (lumin.nn.callbacks.cyclic_callbacks.AbsCyclicCallback method), 29
on_epoch_begin() (lumin.nn.callbacks.cyclic_callbacks.CycleStep method), 32
on_epoch_begin() (lumin.nn.callbacks.model_callbacks.SWA method), 36
on_epoch_begin() (lumin.nn.callbacks.monitors.MetricLogger method), 38
on_epoch_begin() (lumin.nn.callbacks.opt_callbacks.LRFinder method), 39
on_epoch_begin() (lumin.nn.metrics.eval_metric.EvalMetric method), 61
on_epoch_end() (lumin.nn.callbacks.data_callbacks.TargReplace method), 34
on_epoch_end() (lumin.nn.callbacks.model_callbacks.SWA method), 36
on_epoch_end() (lumin.nn.callbacks.monitors.EarlyStopping method), 37
on_epoch_end() (lumin.nn.callbacks.monitors.EpochSaver method), 39
on_epoch_end() (lumin.nn.callbacks.monitors.MetricLogger method), 38
on_epoch_end() (lumin.nn.callbacks.monitors.SaveBest method), 37
on_epoch_end() (lumin.nn.metrics.eval_metric.EvalMetric method), 61
on_fold_begin() (lumin.nn.callbacks.adversarial_callbacks.PivotTraining method), 28
on_fold_begin() (lumin.nn.callbacks.data_callbacks.BinaryLabelSmooth method), 32
on_fold_begin() (lumin.nn.callbacks.data_callbacks.BootstrapResample method), 33
on_fold_begin() (lumin.nn.callbacks.data_callbacks.TargReplace method), 34
on_fold_begin() (lumin.nn.callbacks.lsuv_init.LsuvInit method), 35
on_fold_begin() (lumin.nn.callbacks.monitors.MetricLogger method), 38
on_fold_end() (lumin.nn.callbacks.monitors.MetricLogger method), 38
on_forwards_end() (lumin.nn.callbacks.adversarial_callbacks.PivotTraining method), 28
on_forwards_end() (lumin.nn.callbacks.pred_handlers.PredHandler method), 40
on_forwards_end() (lumin.nn.metrics.eval_metric.EvalMetric method), 61
on_pred_begin() (lumin.nn.callbacks.callback.Callback method), 28
on_pred_begin() (lumin.nn.callbacks.data_callbacks.ParametrisedPrediction method), 33
on_pred_begin() (lumin.nn.callbacks.pred_handlers.PredHandler method), 40
on_pred_end() (lumin.nn.callbacks.pred_handlers.PredHandler method), 40
on_train_begin() (lumin.nn.callbacks.adversarial_callbacks.PivotTraining method), 28
on_train_begin() (lumin.nn.callbacks.callback.Callback method), 28
on_train_begin() (lumin.nn.callbacks.cyclic_callbacks.AbsCyclicCallback method), 29
on_train_begin() (lumin.nn.callbacks.cyclic_callbacks.CycleStep method), 32
on_train_begin() (lumin.nn.callbacks.data_callbacks.BootstrapResample method), 33
on_train_begin() (lumin.nn.callbacks.lsuv_init.LsuvInit method), 35
on_train_begin() (lumin.nn.callbacks.model_callbacks.SWA method), 36
on_train_begin() (lumin.nn.callbacks.monitors.EarlyStopping method), 37

on_train_begin() (*lumin.nn.callbacks.monitors.EpochSaver method*), 39
on_train_begin() (*lumin.nn.callbacks.monitors.MetricLogger method*), 38
on_train_begin() (*lumin.nn.callbacks.monitors.SaveBest method*), 37
on_train_begin() (*lumin.nn.callbacks.opt_callbacks.LRFinder method*), 39
on_train_begin() (*lumin.nn.metrics.eval_metric.EvalMetric method*), 61
on_train_end() (*lumin.nn.callbacks.adversarial_callbacks.PivotTraining method*), 28
on_train_end() (*lumin.nn.callbacks.cyclic_callbacks.CycleStep method*), 32
on_train_end() (*lumin.nn.callbacks.monitors.SaveBest method*), 37
OneCycle (*class in lumin.nn.callbacks.cyclic_callbacks*), 31

P

ParametrisedPrediction (*class in lumin.nn.callbacks.data_callbacks*), 33
PivotTraining (*class in lumin.nn.callbacks.adversarial_callbacks*), 27
plot() (*lumin.nn.callbacks.cyclic_callbacks.AbsCyclicCallback method*), 30
plot() (*lumin.nn.callbacks.cyclic_callbacks.CycleStep method*), 32
plot() (*lumin.nn.callbacks.cyclic_callbacks.OneCycle method*), 31
plot() (*lumin.nn.callbacks.opt_callbacks.LRFinder method*), 39
plot_1d_partial_dependence() (*in module lumin.plotting.interpretation*), 124
plot_2d_partial_dependence() (*in module lumin.plotting.interpretation*), 125
plot_binary_class_pred() (*in module lumin.plotting.results*), 129
plot_binary_sample_feat() (*in module lumin.plotting.data_viewing*), 123
plot_bottleneck_weighted_inputs() (*in module lumin.plotting.interpretation*), 127
plot_embedding() (*in module lumin.plotting.interpretation*), 124
plot_embeds() (*lumin.nn.models.blocks.head.CatEmbHead method*), 81
plot_feat() (*in module lumin.plotting.data_viewing*), 121
plot_importance() (*in module lumin.plotting.interpretation*), 124
plot_kdes_from_bs() (*in module lumin.plotting.data_viewing*), 122
plot_lr() (*lumin.nn.callbacks.opt_callbacks.LRFinder method*), 39
plot_lr_finders() (*in module lumin.plotting.training*), 130
plot_multibody_weighted_outputs() (*in module lumin.plotting.interpretation*), 126
plot_rank_order_dendrogram() (*in module lumin.plotting.data_viewing*), 122
plot_roc() (*in module lumin.plotting.results*), 128
plot_sample_pred() (*in module lumin.plotting.results*), 129
plot_train_history() (*in module lumin.plotting.training*), 130
PlotSettings (*class in lumin.plotting.plot_settings*), 128
PredHandler (*class in lumin.nn.callbacks.pred_handlers*), 40
predict() (*lumin.nn.ensemble.ensemble.Ensemble method*), 50
predict() (*lumin.nn.models.blocks.endcap.AbsEndcap method*), 73
predict() (*lumin.nn.models.model.Model method*), 103
print_losses() (*lumin.nn.callbacks.monitors.MetricLogger method*), 38
proc_cats() (*in module lumin.data_processing.pre_proc*), 20
proc_event() (*in module lumin.data_processing.hep_proc*), 16

R

RecurrentHead (*class in lumin.nn.models.blocks.head*), 84
RegAsProxyPull (*class in lumin.nn.metrics.reg_eval*), 62
RegPull (*class in lumin.nn.metrics.reg_eval*), 61
remove() (*lumin.utils.misc.ForwardHook method*), 135
repeated_rf_rank_features() (*in module lumin.optimisation.features*), 113
Res1DBlock (*class in lumin.nn.models.blocks.conv_blocks*), 68
ResNeXt1DBlock (*class in lumin.nn.models.blocks.conv_blocks*), 69

rf_check_feat_removal() (*in module lumin.optimisation.features*), 112
rf_rank_features() (*in module lumin.optimisation.features*), 111
RocAucScore (*class in lumin.nn.metrics.class_eval*), 58
row_wise (*lumin.nn.models.blocks.gnn_blocks.GravNet attribute*), 79
row_wise (*lumin.nn.models.blocks.gnn_blocks.InteractionNet attribute*), 77
RunningBatchNorm1d (*class in lumin.nn.models.layers.batchnorms*), 95
RunningBatchNorm2d (*class in lumin.nn.models.layers.batchnorms*), 96
RunningBatchNorm3d (*class in lumin.nn.models.layers.batchnorms*), 96

S

save() (*lumin.nn.ensemble.ensemble.Ensemble method*), 50
save() (*lumin.nn.models.model.Model method*), 103
save_embeds() (*lumin.nn.models.blocks.head.CatEmbHead method*), 82
save_fold_pred() (*lumin.nn.data.fold_yielder.FoldYielder method*), 44
save_to_grp() (*in module lumin.data_processing.file_proc*), 11
SaveBest (*class in lumin.nn.callbacks.monitors*), 37
SEBlock1d (*class in lumin.nn.models.blocks.conv_blocks*), 71
SEBlock2d (*class in lumin.nn.models.blocks.conv_blocks*), 72
SEBlock3d (*class in lumin.nn.models.blocks.conv_blocks*), 72
SelfAttention (*class in lumin.nn.models.layers.self_attention*), 97
set_input_mask() (*lumin.nn.models.model.Model method*), 103
set_layers() (*lumin.nn.models.blocks.conv_blocks.Conv1DBlock method*), 68
set_layers() (*lumin.nn.models.blocks.conv_blocks.Res1DBlock method*), 69
set_layers() (*lumin.nn.models.blocks.conv_blocks.ResNeXt1DBlock method*), 70
set_lr() (*lumin.nn.models.model.Model method*), 103
set_lr() (*lumin.nn.models.model_builder.ModelBuilder method*), 107
set_model() (*lumin.nn.callbacks.callback.Callback method*), 28
set_mom() (*lumin.nn.models.model.Model method*), 104
set_plot_settings() (*lumin.nn.callbacks.callback.Callback method*), 29
set_weights() (*lumin.nn.models.model.Model method*), 104
SignificanceLoss (*class in lumin.nn.losses.hep_losses*), 56
str2bool() (*in module lumin.utils.misc*), 134
str2sz() (*lumin.plotting.plot_settings.PlotSettings method*), 128
subsample_df() (*in module lumin.utils.misc*), 135
SWA (*class in lumin.nn.callbacks.model_callbacks*), 35
Swish (*class in lumin.nn.models.layers.activations*), 95

T

TargReplace (*class in lumin.nn.callbacks.data_callbacks*), 33
to_binary_class() (*in module lumin.utils.misc*), 134
to_cartesian() (*in module lumin.data_processing.hep_proc*), 14
to_device() (*in module lumin.utils.misc*), 133
to_np() (*in module lumin.utils.misc*), 133
to_pt_eta_phi() (*in module lumin.data_processing.hep_proc*), 14
to_tensor() (*in module lumin.utils.misc*), 133
train_models() (*in module lumin.nn.training.train*), 108
twist() (*in module lumin.data_processing.hep_proc*), 14

U

uncert_round() (*in module lumin.utils.statistics*), 136
unfreeze_layers() (*lumin.nn.models.model.Model method*), 104
update_plot() (*lumin.nn.callbacks.monitors.MetricLogger method*), 39
update_stats() (*lumin.nn.models.layers.batchnorms.RunningBatchNorm1d method*), 96

W

WeightedBinnedHuber (*class in lumin.nn.losses.advanced_losses*), 53
WeightedCCE (*class in lumin.nn.losses.basic_weighted*), 55
WeightedFractionalBinnedHuber (*class in lumin.nn.losses.advanced_losses*), 53
WeightedFractionalMSE (*class in lumin.nn.losses.advanced_losses*), 52
WeightedMAE (*class in lumin.nn.losses.basic_weighted*), 55

WeightedMSE (*class in lumin.nn.losses.basic_weighted*), 54