LUMIN

LUMIN Contributors

Nov 15, 2019

PACKAGE REFERENCE

Lumin Unifies Many Improvements for Networks

LUMIN aims to become 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

LUMIN.DATA_PROCESSING PACKAGE

1.1 Submodules

1.2 lumin.data_processing.file_proc module

lumin.data_processing.file_proc.**save_to_grp**(*arr*, *grp*, *name*)

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

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*)

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** (*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*) – any extra columns to save
- **wgt_feat** (*optional*) – column to save as data weights

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*)

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** (*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*) – column to use for stratified splitting
- **misc_feats** (*optional*) – any extra columns to save
- **wgt_feat** (*optional*) – column to save as data weights

1.3 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.array

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.array

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_z(), 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.array 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[<built-in function array>, 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 <built-in function array>

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[<built-in function array>, 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[<built-in function array>, 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, <built-in function array>]

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.array

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[<built-in function array>, 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[<built-in function array>, 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[<built-in function array>, 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, <built-in function array>]

Returns array of boosted deltaR in not inplace

1.4 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 categories to codes
ordered dictionary mapping categorical features to their cardinalities

1.5 Module contents

CHAPTER

TWO

LUMIN.EVALUATION PACKAGE

2.1 Submodules

2.2 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 accross 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 accross 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

2.3 Module contents

CHAPTER

THREE

LUMIN.INFERENCE PACKAGE

3.1 Submodules

3.2 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

3.3 Module contents

CHAPTER

FOUR

LUMIN.NN PACKAGE

4.1 Subpackages

4.1.1 lumin.nn.callbacks package

Submodules

lumin.nn.callbacks.callback module

class lumin.nn.callbacks.callback.**Callback**(*model=None*, *plot_settings=<lumin.plotting.plot_settings.PlotSettings object>*)

Bases: lumin.nn.callbacks.abs_callback.AbsCallback

Base callback class from which other callbacks should inherit.

Parameters

- **model** (Optional[AbsModel]) – model to refer to during training
- **plot_settings** (*PlotSettings*) – PlotSettings class

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*, *model=None*, *nb=None*, *plot_settings=<lumin.plotting.plot_settings.P object>*)

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.
- **model** (Optional[AbsModel]) – model to refer to during training
- **nb** (Optional[int]) – number of minibatches (iterations) to expect per epoch
- **plot_settings** (*PlotSettings*) – PlotSettings class

on_batch_begin(***kargs*)
Computes the new value for the optimiser parameter and returns it

Return type float

Returns new value for optimiser parameter

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

Return type None

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

Return type None

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

Return type None

set_nb(*nb*)
Sets the callback's internal number of iterations per cycle equal to *nb*scale*

Parameters **nb** (int) – number of minibatches per epoch

Return type None

class lumin.nn.callbacks.cyclic_callbacks.**CycleLR**(*lr_range*, *interp='cosine'*, *cycle_mult=1*, *decrease_param='auto'*, *scale=1*, *model=None*, *nb=None*, *plot_settings=<lumin.plotting.plot_settings.PlotSettings object>*)

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 (sub-)epoch
- **model** (Optional[AbsModel]) – *Model* to alter, alternatively call `set_model()`.
- **nb** (Optional[int]) – Number of batches in a (sub-)epoch
- **plot_settings** (*PlotSettings*) – *PlotSettings* class to control figure appearance

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)
```

on_batch_begin(***kargs*)

Computes the new lr and assignes it to the optimiser

Return type None

class lumin.nn.callbacks.cyclic_callbacks.**CycleMom**(*mom_range*, *interp='cosine'*, *cycle_mult=1*, *decrease_param='auto'*, *scale=1*, *model=None*, *nb=None*, *plot_settings=<lumin.plotting.plot_settings.PlotSettings object>*)

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 (sub-)epoch
- **model** (Optional[AbsModel]) – *Model* to alter, alternatively call set_model()
- **nb** (Optional[int]) – Number of batches in a (sub-)epoch
- **plot_settings** (*PlotSettings*) – *PlotSettings* class to control figure appearance

Examples::

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

on_batch_begin(***kargs*)

Computes the new momentum and assignes it to the optimiser

Return type None

class lumin.nn.callbacks.cyclic_callbacks.**OneCycle**(*lengths*, *lr_range*, *mom_range=(0.85, 0.95)*, *interp='cosine'*, *model=None*, *nb=None*, *plot_settings=<lumin.plotting.plot_settings.PlotSettings object>*)

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 (sub-)epochs in first and second stages of cycle
- **lr_range** (List[float]) – tuple of initial and final LRs
- **mom_range** (Tuple[float, float]) – tuple of initial and final momenta
- **interp** (str) – 'cosine' or 'linear' interpolation
- **model** (Optional[AbsModel]) – *Model* to alter, alternatively call set_model()
- **nb** (Optional[int]) – Number of batches in a (sub-)epoch
- **plot_settings** (*PlotSettings*) – *PlotSettings* class to control figure appearance

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(***kargs*)
Computes the new lr and momentum and assignes them to the optimiser

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*, *model=None*)
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 and inference.

Parameters

- **coefs** (Union[float, Tuple[float, float]]) – Smoothing coefficients: 0->coef[0] 1->1-coef[1]. if passed float, coef[0]=coef[1]
- **model** (Optional[AbsModel]) – not used, only for compatability

Examples::

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

on_epoch_begin(*by*, ***kargs*)
Apply smoothing at train-time

Return type None

on_eval_begin(*targets*, ***kargs*)
Apply smoothing at test-time

Return type None

class lumin.nn.callbacks.data_callbacks.**SequentialReweight**(*reweight_func*, *scale=0.1*, *model=None*)
Bases: *lumin.nn.callbacks.callback.Callback*

Caution: Experiemntal proceedure

During ensemble training, sequentially reweight training data in last validation fold based on prediction performance of last trained model. Reweighting highlights data which are easier or more difficult to predict to the next model being trained.

Parameters

- **reweight_func** (Callable[[Tensor, Tensor], Tensor]) – callable function returning a tensor of same shape as targets, ideally quantifying model-prediction performance
- **scale** (float) – multiplicative factor for rescaling returned tensor of reweight_func
- **model** (Optional[AbsModel]) – *Model* to provide predictions, alternatively call set_model()

Examples::

```
>>> seq_reweight = SequentialReweight(
...     reweight_func=nn.BCELoss(reduction='none'), scale=0.1)
```

on_train_end(*fy*, *val_id*, ***kargs*)

Reweighs the validation fold once training is finished

Parameters

- **fy** (*FoldYielder*) – FoldYielder providing the training and validation data
- **fold_id** – Fold index which was used for validation

Return type None

class lumin.nn.callbacks.data_callbacks.**SequentialReweightClasses**(*reweight_func*, *scale=0.1*, *model=None*)

Bases: *lumin.nn.callbacks.data_callbacks.SequentialReweight*

Caution: Experiemntal proceedure

Version of *SequentialReweight* designed for classification, which renormalises class weights to original weight-sum after reweighting During ensemble training, sequentially reweight training data in last validation fold based on prediction performance of last trained model. Reweighting highlights data which are easier or more difficult to predict to the next model being trained.

Parameters

- **reweight_func** (Callable[[Tensor, Tensor], Tensor]) – callable function returning a tensor of same shape as targets, ideally quantifying model-prediction performance
- **scale** (float) – multiplicative factor for rescaling returned tensor of reweight_func
- **model** (Optional[AbsModel]) – *Model* to provide predictions, alternatively call set_model()

Examples::

```
>>> seq_reweight = SequentialReweight(
...     reweight_func=nn.BCELoss(reduction='none'), scale=0.1)
```

class lumin.nn.callbacks.data_callbacks.**BootstrapResample**(*n_folds*, *bag_each_time=False*, *reweight=True*, *model=None*)

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

- **model** (Optional[AbsModel]) – not used, only for compatability

Examples::

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

on_epoch_begin(*by*, ***kargs*)
Resamples training data for new epoch

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

Return type None

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

Return type None

class lumin.nn.callbacks.data_callbacks.**FeatureSubsample**(*cont_feats*, *model=None*)
Bases: *lumin.nn.callbacks.callback.Callback*

Callback for training a model on a random sub-sample of the range of possible input features. Only sub-samples continuous features. Number of continuous inputs infered from model. Associated *Model* will automatically mask its inputs during inference; simply provide inputs with the same number of columns as trainig data.

Attention: This callback is now depreciated in favour of passing *cont_subsample_rate* and *guaranteed_feats* to *ModelBuilder* as these offer greater functionality and are compatable with using a MultiBlock body. Will be removed in *V0.5*.

Caution: This callback is incompatable with using a MultiBlock body

Parameters

- **cont_feats** (List[str]) – list of all continuous features in input data. Order must match.
- **model** (Optional[AbsModel]) – *Model* being trained, alternatively call set_model()

Examples::

```
>>> feat_subsample = FeatureSubsample(cont_feats=['pT', 'eta', 'phi'])
```

on_train_begin(***kargs*)
Subsamples features for use in training and sets model's input mask for inference

Return type None

lumin.nn.callbacks.loss_callbacks module

class lumin.nn.callbacks.loss_callbacks.**GradClip**(*clip*, *clip_norm=True*, *model=None*)
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_*)
- **model** (Optional[AbsModel]) – *Model* with parameters to clip gradients, alternatively call set_model()

Examples::

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

on_backwards_end(***kargs*)
Clips gradients prior to parameter updates

Return type None

lumin.nn.callbacks.model_callbacks module

class lumin.nn.callbacks.model_callbacks.**SWA**(*start_epoch*, *renewal_period=-1*, *model=None*, *val_fold=None*, *cyclic_callback=None*, *update_on_cycle_end=None*, *verbose=False*, *plot_settings=<lumin.plotting.plot_settings.PlotSettings object>*)
Bases: *lumin.nn.callbacks.model_callbacks.AbsModelCallback*

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 x0 will begin to be tracked start_epoch (sub-)epochs/cycles after training begins.
- *cycle_since_replacement* is set to 1
- Renewal_period (sub-)epochs/cycles later, a second average x1 will be tracked.
- At the next renewal period, the performance of x0 and x1 will be compared on data contained in val_fold.
  - **If x0 is better than x1:**
    - x1 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:**
    - x0 is replaced by x1
    - x1 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) – (sub-)epoch/cycle to begin averaging

- **renewal_period** (int) – How often to check performance of averages, and renew tracking of least performant
- **model** (Optional[AbsModel]) – *Model* to provide parameters, alternatively call set_model()
- **val_fold** (Optional[Dict[str, ndarray]]) – Dictionary containing inputs, targets, and weights (or None) as Numpy arrays
- **cyclic_callback** (Optional[*AbsCyclicCallback*]) – Optional for any cyclical callback which is running
- **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
- **plot_settings** (*PlotSettings*) – *PlotSettings* class to control figure appearance

Examples::

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

get_loss()
Evaluates SWA model and returns loss

Return type float

Returns Loss on validation fold for oldest SWA average

on_epoch_begin(***kargs*)
Resets loss to prepare for new epoch

Return type None

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

Return type None

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

Return type None

class lumin.nn.callbacks.model_callbacks.**AbsModelCallback**(*model=None*, *val_fold=None*, *cyclic_callback=None*, *update_on_cycle_end=None*, *plot_settings=<lumin.plotting.plot_settings.Plot object>*)

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

Abstract class for callbacks which provide alternative models during training

Parameters

- **model** (Optional[AbsModel]) – *Model* to provide parameters, alternatively call set_model()

- **val_fold** (Optional[Dict[str, ndarray]]) – Dictionary containing inputs, targets, and weights (or None) as Numpy arrays
- **cyclic_callback** (Optional[*AbsCyclicCallback*]) – Optional for any cyclical callback which is running
- **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.
- **plot_settings** (*PlotSettings*) – *PlotSettings* class to control figure appearance

abstract get_loss()

Return type float

set_cyclic_callback(*cyclic_callback*)
Sets the cyclical callback to lock into for updating new models

Return type None

set_val_fold(*val_fold*)
Sets the validation fold used for evaluating new models

Return type None

lumin.nn.callbacks.opt_callbacks module

class lumin.nn.callbacks.opt_callbacks.**LRFinder**(*nb*, *lr_bounds=[1e-07, 10]*, *model=None*, *plot_settings=<lumin.plotting.plot_settings.PlotSettings object>*)
Bases: *lumin.nn.callbacks.callback.Callback*

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

Parameters

- **nb** (int) – number of batches in a (sub-)epoch
- **lr_bounds** (Tuple[float, float]) – tuple of initial and final LR
- **model** (Optional[AbsModel]) – Model to alter, alternatively call set_model()
- **plot_settings** (*PlotSettings*) – *PlotSettings* class to control figure appearance

get_df()
Returns a DataFrame of LRs and losses

Return type DataFrame

on_batch_end(*loss*, ***kargs*)
Records loss and increments LR

Parameters **loss** (float) – training loss for most recent batch

Return type None

on_train_begin(***kargs*)
Prepares variables and optimiser for new training

Return type None

plot(*n_skip=0*, *n_max=None*, *lim_y=None*)
Plot the loss as a function of the LR.

Parameters

- **n_skip** (int) – Number of initial iterations to skip in plotting
- **n_max** (Optional[int]) – Maximum iteration number to plot
- **lim_y** (Optional[Tuple[float, float]]) – y-range for plotting

Return type None

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

Return type None

Module contents

4.1.2 lumin.nn.data package

Submodules

lumin.nn.data.batch_yielder module

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

Bases: object

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

Parameters

- **inputs** (ndarray) – input array for (sub-)epoch
- **targets** (ndarray) – targte 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** – 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** – 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.

lumin.nn.data.fold_yielder module

class lumin.nn.data.fold_yielder.**FoldYielder**(*foldfile*, *cont_feats*, *cat_feats*, *ignore_feats=None*, *input_pipe=None*, *output_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** (List[str]) – list of names of continuous features present in input data
- **cat_feats** (List[str]) – list of names of categorical features present in input data
- **ignore_feats** (Optional[List[str]]) – optional list of input features which should be ignored
- **input_pipe** (Union[str, Pipeline, None]) – optional Pipeline, or filename for pickled Pipeline, which was used for processing the inputs
- **output_pipe** (Union[str, Pipeline, None]) – optional Pipeline, or filename for pickled Pipeline, which was used for processing the targets

Examples::

```
>>> fy = FoldYielder('train.h5', cont_feats=['pT','eta','phi','mass'],
...                  cat_feats=['channel'], ignore_feats=['phi'],
...                  input_pipe='input_pipe.pkl')
```

add_ignore(*feats*)
Add features to ignored features.

Parameters **feats** (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** (Pipeline) – Pipeline which was used for preprocessing the input data

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

Return type None

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

Parameters **output_pipe** (Pipeline) – Pipeline which was used for preprocessing the target data

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** (str) – 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 accounts 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_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*)
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

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 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

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

set_foldfile(*foldfile*)

Sets the file from which to access data

Parameters **foldfile** (Union[str, Path, File]) – filname of h5py file or opened h5py file

Return type None

class lumin.nn.data.fold_yielder.**HEPAugFoldYielder**(*foldfile*, *cont_feats*, *cat_feats*, *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*)

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** (List[str]) – list of names of continuous features present in input data
- **cat_feats** (List[str]) – list of names of categorical features present in input data
- **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

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 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 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

4.1.3 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.

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_ouput_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

build_ensemble(*results*, *size*, *model_builder*, *metric='loss'*, *weighting='reciprocal'*, *higher_metric_better=False*, *snapshot_args=None*, *location=PosixPath('train_weights')*, *verbose=True*)

Load up an instantiated *Ensemble* with outputs of *fold_train_ensemble()*

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
- **location** (Path) – Path to save location passed to *fold_train_ensemble()*
- **verbose** (bool) – whether to print out information of models loaded

Examples::

```
>>> ensemble.build_ensemble(results, 10, model_builder,
...                         location=Path('train_weights'))
>>>
>>> ensemble.build_ensemble(
...     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})
```

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_results(*results*, *size*, *model_builder*, *metric='loss'*, *weighting='reciprocal'*, *higher_metric_better=False*, *snapshot_args=None*, *location=PosixPath('train_weights')*, *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
- **location** (Path) – Path to save location passed to *fold_train_ensemble()*
- **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(
```

(continues on next page)

(continued from previous page)

```
...     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*, *eval_metric=None*)

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

Parameters

- **fy** (*FoldYielder*) – *FoldYielder* interfacing to data on which to evaluate importance
- **eval_metric** (Optional[*EvalMetric*]) – Optional EvalMetric to use for quantifying performance

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'*, *callbacks=None*, *verbose=True*)

Compatability method for predicting data contained in either a Numpy array or a *FoldYielder*

Will either pass inputs to *lumin.nn.ensemble.ensemble.Ensemble.predict_array()* or *lumin.nn.ensemble.ensemble.Ensemble.predict_folds()*.

Parameters

- **inputs** (Union[ndarray, *FoldYielder*, List[ndarray]]) – either a *FoldYielder* interfacing with the input data, or the input data as an array
- **n_models** (Optional[int]) – number of models to use in predictions as ranked by the metric which was used when constructing the *Ensemble*. By default, entire ensemble is used.
- **pred_name** (str) – name for new group of predictions if passed a *FoldYielder*
- **callbacks** (Optional[List[AbsCallback]]) – list of any callbacks to use during evaluation
- **verbose** (bool) – whether to print average predicition timings

Return type Union[None, ndarray]

Returns If passed a Numpy array will return predictions.

Examples::

```
>>> preds = ensemble.predict(input_array)
>>>
>>> ensemble.predict(test_fy)
```

predict_array(*arr*, *n_models=None*, *parent_bar=None*, *display=True*, *callbacks=None*)

Apply ensemble to Numpy array and get predictions. If an output pipe has been added to the ensemble, then the predictions will be deprocessed. Inputs are expected to be preprocessed; i.e. any input pipe added to the ensemble is not used.

Parameters

- **arr** (ndarray) – input data
- **n_models** (Optional[int]) – number of models to use in predictions as ranked by the metric which was used when constructing the *Ensemble*. By default, entire ensemble is used.
- **parent_bar** (Optional[ConsoleMasterBar]) – not used when calling the method directly
- **display** (bool) – whether to display a progress bar for model evaluations
- **callbacks** (Optional[List[AbsCallback]]) – list of any callbacks to use during evaluation

Return type ndarray

Returns Numpy array of predictions

Examples::

```
>>> preds = ensemble.predict_array(inputs)
```

predict_folds(*fy*, *n_models=None*, *pred_name='pred'*, *callbacks=None*, *verbose=True*)

Apply ensemble to data accessed by a *FoldYielder* and save predictions as a new group per fold in the foldfile. If an output pipe has been added to the ensemble, then the predictions will be deprocessed. Inputs

are expected to be preprocessed; i.e. any input pipe added to the ensemble is not used. If foldyielder has test-time augmentation, then predictions will be averaged over all augmentated forms of the data.

Parameters

- **fy** (*FoldYielder*) – *FoldYielder* interfacing with the input data
- **n_models** (Optional[int]) – number of models to use in predictions as ranked by the metric which was used when constructing the *Ensemble*. By default, entire ensemble is used.
- **pred_name** (str) – name for new group of predictions
- **callbacks** (Optional[List[AbsCallback]]) – list of any callbacks to use during evaluation
- **verbose** (bool) – whether to print average prediction timings

Examples::

```
>>> ensemble.predict_array(test_fy, pred_name='pred_tta')
```

Return type None

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

4.1.4 lumin.nn.interpretation package

Submodules

lumin.nn.interpretation.features module

lumin.nn.interpretation.features.**get_nn_feat_importance**(*model*, *fy*, *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
- **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) – whetehr 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*, *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

- **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))
```

Module contents

4.1.5 lumin.nn.losses package

Submodules

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

4.1.6 lumin.nn.metrics package

Submodules

lumin.nn.metrics.class_eval module

class lumin.nn.metrics.class_eval.**AMS**(*n_total*, *wgt_name*, *targ_name='targets'*, *br=0*, *syst_unc_b=0*, *use_quick_scan=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.
- **targ_name** (str) – name of target group in fold file
- **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)

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(*fy*, *idx*, *y_pred*)

Compute maximum AMS on fold using provided predictions.

Parameters

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

Return type float

Returns Maximum AMS computed on reweighted data from fold

Examples::

```
>>> ams = ams_metric.evaluate(train_fy, val_id, val_preds)
```

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*)

Bases: *lumin.nn.metrics.class_eval.AMS*

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)

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(*fy*, *idx*, *y_pred*)

Compute maximum AMS on fold using provided predictions.

Parameters

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

Return type float

Returns Maximum AMS computed on reweighted data from fold

Examples::

```
>>> ams = ams_metric.evaluate(train_fy, val_id, val_preds)
```

lumin.nn.metrics.eval_metric module

class lumin.nn.metrics.eval_metric.**EvalMetric**(*targ_name*, *wgt_name=None*)

Bases: abc.ABC

Abstract class for evaluating performance of a model using some metric

Parameters

- **targ_name** (str) – name of group in fold file containing regression targets

- **wgt_name** (Optional[str]) – name of group in fold file containing datapoint weights

abstract evaluate(*fy*, *idx*, *y_pred*)
Evaluate the required metric for a given fold and set of predictions

Parameters

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

Return type float

Returns metric value

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

Parameters

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

Return type DataFrame

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

lumin.nn.metrics.reg_eval module

class lumin.nn.metrics.reg_eval.**RegPull**(*return_mean*, *use_bootstrap=False*, *use_weights=True*, *use_pull=True*, *targ_name='targets'*, *wgt_name=None*)
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_weights** (bool) – 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** (str) – name of group in fold file containing regression targets
- **wgt_name** (Optional[str]) – name of group in fold file containing datapoint weights

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)
```

(continues on next page)

(continued from previous page)

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

evaluate (*fy*, *idx*, *y_pred*)

Compute statisitic on fold using provided predictions.

Parameters

- **fy** (*FoldYielder*) – *FoldYielder* interfacing to data
- **idx** (int) – fold index corresponding to fold for which y_pred was computed
- **y_pred** (ndarray) – 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)
```

class lumin.nn.metrics.reg_eval.**RegAsProxyPull** (*proxy_func*, *return_mean*, *use_bootstrap=False*, *use_weights=True*, *use_pull=True*, *targ_name='targets'*, *wgt_name=None*)

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** (bool) – 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** (str) – name of group in fold file containing regression targets
- **wgt_name** (Optional[str]) – name of group in fold file containing datapoint weights

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
...                                 in enumerate(targ_feats)})
>>>     df['gen_target'] = 125
```

(continues on next page)

(continued from previous page)

```
>>>
>>> std_delta = RegAsProxyPull(proxy_func=reg_proxy_func,
...                            return_mean=False, use_pull=False)
```

evaluate (*fy*, *idx*, *y_pred*)

Compute statisitic on fold using provided predictions.

Parameters

- **fy** (*FoldYielder*) – *FoldYielder* interfacing to data
- **idx** (int) – fold index corresponding to fold for which y_pred was computed
- **y_pred** (ndarray) – 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

4.1.7 lumin.nn.models package

Subpackages

lumin.nn.models.blocks package

Submodules

lumin.nn.models.blocks.body module

class lumin.nn.models.blocks.body.**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*)

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

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 body

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

(continues on next page)

(continued from previous page)

```
...     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 body

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

Returns Resulting tensor

Return type Tensor

get_out_size()

Get size width of output layer

Return type int

Returns Total number of outputs accross all blocks

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.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)
```

(continues on next page)

(continued from previous page)

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

forward(*x_in*)
Pass tensor through head

Parameters **x** – 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

lumin.nn.models.blocks.tail module

class lumin.nn.models.blocks.tail.**ClassRegMulti**(*n_in*, *n_out*, *objective*, *y_range=None*, *bias_init=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.
- **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 tail

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

Returns Resulting tensor of model outputs

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

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.

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

Examples::

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

evaluate(*inputs*, *targets*, *weights=None*, *callbacks=None*, *mask_inputs=True*)

Compute loss on provided data.

Parameters

- **inputs** (Tensor) – input data as tensor on device
- **targets** (Tensor) – targets as tensor on device
- **weights** (Optional[Tensor]) – Optional weights as tensor on device
- **callbacks** (Optional[List[AbsCallback]]) – list of any callbacks to use during evaluation
- **mask_inputs** (bool) – whether to apply input mask if one has been set

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(*batch_yielder*, *callbacks=None*, *mask_inputs=True*)
Fit network for one complete iteration of a *BatchYielder*, i.e. one (sub-)epoch

Parameters

- **batch_yielder** (*BatchYielder*) – *BatchYielder* providing training data in form of tuple of inputs, targtes, and weights as tensors on device
- **callbacks** (Optional[List[AbsCallback]]) – list of AbsCallback to be used during training
- **mask_inputs** (bool) – whether to apply input mask if one has been set

Return type float

Returns Loss on training data averaged across all minibatches

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*, *eval_metric=None*)
Call *get_nn_feat_importance()* passing this *Model* and provided arguments

Parameters

- **fy** (*FoldYielder*) – *FoldYielder* interfacing to data on which to evaluate importance
- **eval_metric** (Optional[*EvalMetric*]) – Optional EvalMetric to use for quantifying performance

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'*, *callbacks=None*, *verbose=True*)
Apply model to inputed data and compute predictions. A compatability method to call *predict_array()* or meth:~*lumin.nn.models.model.Model.predict_folds*, depending on input type.

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*
- **callbacks** (Optional[List[AbsCallback]]) – list of any callbacks to use during evaluation

- **verbose** (bool) – whether to print average prediction timings

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

predict_array(*inputs*, *as_np=True*, *mask_inputs=True*, *callbacks=None*)

Pass inputs through network and obtain predictions.

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)
- **mask_inputs** (bool) – whether to apply input mask if one has been set
- **callbacks** (Optional[List[AbsCallback]]) – list of any callbacks to use during evaluation

Return type Union[ndarray, Tensor]

Returns Model prediction(s) per datapoint

predict_folds(*fy*, *pred_name='pred'*, *callbacks=None*, *verbose=True*)

Apply model to all dataaccessed by a *FoldYielder* and save predictions as new group in fold file

Parameters

- **fy** (*FoldYielder*) – *FoldYielder* interfacing to data
- **pred_name** (str) – name of group to which to save predictions
- **callbacks** (Optional[List[AbsCallback]]) – list of any callbacks to use during evaluation
- **verbose** (bool) – whether to print average prediction timings

Return type None

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

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), RAdam, Ranger, 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** (AbsHead) – uninstantiated class which can receive input data and upscale it to model width
- **body** (AbsBody) – uninstantiated class which implements the main bulk of the model's hidden layers
- **tail** (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,
>>>                                                  '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,
```

(continues on next page)

(continued from previous page)

```
>>>                                                 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,
```

(continues on next page)

(continued from previous page)

```
...                 '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, Any]

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

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

4.1.8 lumin.nn.training package

Submodules

lumin.nn.training.fold_train module

lumin.nn.training.fold_train.**fold_train_ensemble**(*fy, n_models, bs, model_builder, callback_partials=None, eval_metrics=None, train_on_weights=True, eval_on_weights=True, patience=10, max_epochs=200, plots=['history', 'realtime'], shuffle_fold=True, shuffle_folds=True, bulk_move=True, savepath=PosixPath('train_weights'), verbose=False, log_output=False, 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 save 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. Training folds are loaded iteratively, and model evaluation takes place after each fold use (a sub-epoch), rather than after ever use of all folds (epoch). Training continues until:

- All of the training folds are used max_epoch number of times;
- Or validation loss does not decrease for patience number of training folds; (or cycles, if using an *AbsCyclicCallback*);
- Or a callback triggers trainign to stop, e.g. *OneCycle*

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
- **callback_partials** (Optional[List[partial]]) – optional list of functools.partial, each of which will a instantiate *Callback* when called
- **eval_metrics** (Optional[Dict[str, *EvalMetric*]]) – list of instantiated EvalMetric. At the end of training, validation data and model predictions will be passed to each, and the results printed and saved
- **train_on_weights** (bool) – If weights are present in training data, whether to pass them to the loss function during training
- **eval_on_weights** (bool) – If weights are present in validation data, whether to pass them to the loss function during validation
- **patience** (int) – number of folds (sub-epochs) or cycles to train without decrease in validation loss before ending training (early stopping)
- **max_epochs** (int) – maximum number of epochs for which to train
- **plots** (List[str]) – list of string representation of plots to produce. currently: 'history': loss history of all models after all training has finished 'realtime': live loss evolution during training 'cycle": call the plot method of the last (if any) *AbsCyclicCallback* listed in callback_partials after every complete model training.

- **shuffle_fold** (bool) – whether to tell *BatchYielder* to shuffle data
- **shuffle_folds** (bool) – whether to shuffle the order of the trainign folds
- **bulk_move** (bool) – whether to pass all training data to device at once, or by minibatch. Bulk moving will be quicker, but may not fit in memory.
- **savepath** (Path) – path to to which to save model weights and results
- **verbose** (bool) – whether to print out extra information during training
- **log_output** (bool) – whether to save printed results to a log file rather than printing them
- **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, list of validation losses at the end of each cycle, ordered by model training. Can be passed to *Ensemble*.

Module contents

4.2 Module contents

CHAPTER

FIVE

LUMIN.OPTIMISATION PACKAGE

5.1 Submodules

5.2 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** (ForestRegressor) – 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. Pairs with a correlation coefficient greater than a set threshold are candidates for removal. Candidate pairs are tested, in order of decreasing correlation, by computing the mean performance of a Random Forests trained on: all remaining training features; all remaining training features except the first feature in the pair; and all remaining training features except the second feature in the pair. If the RF trained on all remaining features consistently outperforms the other two trainings, then neither feature from the pair is removed, otherwise the feature whose removal causes the largest mean increase in performance is removed.

Since multiple features maybe correlated with one-another, but this function examines paris of features, it might be necessary/desirable to rerun it on the the previous results.

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 subsample 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 retunred 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

5.3 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.**fold_lr_find**(*fy*, *model_builder*, *bs*, *train_on_weights=True*, *shuffle_fold=True*, *n_folds=-1*, *lr_bounds=[1e-05, 10]*, *callback_partials=None*, *plot_settings=<lumin.plotting.plot_settings.PlotSettings object>*)

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 1 fold, 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
- **train_on_weights** (bool) – If weights are present, whether to use them for training
- **shuffle_fold** (bool) – whether to shuffle data in folds
- **n_folds** (int) – 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
- **callback_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

Return type List[*LRFinder*]

Returns List of *LRFinder* which were used for each model trained

5.4 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

5.5 Module contents

CHAPTER

SIX

LUMIN.PLOTTING PACKAGE

6.1 Submodules

6.2 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': 'Density'}*, *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.

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

- **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*)

Plot 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>*)

Plot dendrogram of features in df clustered via Spearman's rank correlation coefficient. Also returns a list pairs 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 List[List[str]]

Returns List of pairs of features with correlation coefficients greater than the threshold

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>*)

Plot 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

6.3 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

6.4 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

6.5 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 advanceed plotter for prediction distirbution 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

6.6 lumin.plotting.training module

lumin.plotting.training.**plot_train_history**(*histories*, *savename=None*, *ignore_trn=True*, *settings=<lumin.plotting.plot_settings.PlotSettings object>*)

Plot histories object returned by *fold_train_ensemble()* showing the loss evolution over time per model trained.

Parameters

- **histories** (List[Dict[str, List[float]]]) – 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** – whether to ignore training loss
- **settings** (*PlotSettings*) – *PlotSettings* class to control figure appearance

Return type None

lumin.plotting.training.**plot_lr_finders**(*lr_finders*, *lr_range=None*, *loss_range='auto'*, *settings=<lumin.plotting.plot_settings.PlotSettings object>*)

Plot mean loss evolution against learning rate for several fold_lr_find.

Parameters

- **lr_finders** (List[*LRFinder*]) – 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

- **settings** (*PlotSettings*) – *PlotSettings* class to control figure appearance

Return type None

6.7 Module contents

CHAPTER

SEVEN

LUMIN.UTILS PACKAGE

7.1 Submodules

7.2 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

7.3 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.**FowardHook**(*module*, *hook_fn=None*)

Bases: object

Create a hook for performing an action based on the forward pass thorugh a nn.Module

Parameters

- **module** – nn.Module to hook
- **hook_fn** – 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

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

7.4 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

7.5 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 0x7f18d2ec75f8>>]) – 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

7.6 Module contents

CHAPTER

EIGHT

PACKAGE DESCRIPTION

For an introduction and motivation for LUMIN, checkout this talk from IML-2019 at CERN: video, slides.

8.1 Distinguishing Characteristics

8.1.1 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`.

8.1.2 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

- 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)
- HEP-specific losses, e.g. Asimov loss Elwood & Krücker, 2018
- 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
- Keras-style callbacks

8.1.3 Feature selection methods

- Dendrograms
- Feature importance via auto-optimised SK-Learn random forests

8.1.4 Interpretation

- Feature importance for models and ensembles
- Embedding visualisation
- 1D & 2D partial dependency plots (via PDPbox)

8.1.5 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

8.1.6 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

8.1.7 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

8.1.8 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

8.1.9 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.*)

8.2 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`

8.2.1 From PyPI

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

Full functionality requires two additional packages as described below.

8.2.2 From source

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

Optionally, run pip install with `-e` flag for development installation. Full functionality requires an additional package as described below.

8.2.3 Additional modules

Full use of LUMIN requires the latest version of PDPbox, but this is not released yet on PyPI, so you'll need to install it from source, too:

- `git clone https://github.com/SauceCat/PDPbox.git && cd PDPbox && pip install -e .` note the `-e` flag to make sure the version number gets set properly.

8.3 Notes

8.3.1 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 provides 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.

8.3.2 Data types

LUMIN is primarily designed for use on columnar data. 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.

8.3.3 Future

The current priority is to imporve the documentation, add unit tests, and expand the examples.

The next step will be to try and increase the user base and number of contributors. I'm aiming to achieve this through presentations, tutorials, blog posts, and papers.

Further improvments 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.

8.3.4 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.

8.3.5 Code style

Nope, the majority of the codebase 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

8.3.6 Why is LUMIN called LUMIN?

Aside from being a recursive accronym (and therefore the best kind of accronym) 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.

8.3.7 Who develops LUMIN

Currently just me - Giles Strong; a British-born, Lisbon-based, PhD student in particle physics at IST, researcher at LIP-Lisbon, member of Marie Curie ITN AMVA4NewPhysics and the CMS collaboration.

Certainly more developers and contributors are welcome to join and help out!

8.3.8 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

NINE

INDICES AND TABLES

- genindex
- modindex

PYTHON MODULE INDEX

INDEX

D

E

F

G

H

I

L

M

O

S

T

U

W