mdlearn

Release 0.0.10a1

Alexander Brace, Heng Ma, Austin Clyde, Debsindhu Bhowmik, C

Aug 02, 2022

CONTENTS

mdlearn: Machine learning for molecular dynamics

Release 0.0.10a1

Date Aug 02, 2022

CHAPTER

ONE

OVERVIEW OF MDLEARN

mdlearn is a Python package for analyzing molecular dynamics simulations machine learning.

CHAPTER

TWO

GETTING INVOLVED

Please ask **questions** or report **bugs** or **enhancement requests** through the Issue Tracker.

CHAPTER

THREE

INSTALLING MDLEARN

Please see the README file for installation instructions and usage.

3.1 API

mdlearn

3.1.1 mdlearn

Modules

mdlearn.data	
mdlearn.metrics	
mdlearn.nn	
mdlearn.utils	Configurations and utilities for model building and training.
mdlearn.visualize	Functions to visualize modeling results.

mdlearn.data

Modules

mdlearn.data.datasets	
mdlearn.data.preprocess	
mdlearn.data.utils	Utility functions for handling PyTorch data objects.

mdlearn.data.datasets

Modules

mdlearn.data.datasets.contact_map	ContactMap Dataset.
mdlearn.data.datasets.feature_vector	
mdlearn.data.datasets.point_cloud	PointCloud Dataset.
mdlearn.data.datasets.time_contact_map	ContactMapTimeSeriesDataset Dataset.

mdlearn.data.datasets.contact_map

ContactMap Dataset.

Classes

ContactMapDataset(*args, **kwargs)	PyTorch Dataset class which stores sparse contact matrix data in memory.
ContactMapHDF5Dataset(*args, **kwargs)	PyTorch Dataset class to load contact matrix data from HDF5 format.

class mdlearn.data.datasets.contact_map.**ContactMapDataset**(**args: Any*, ***kwargs: Any*)

PyTorch Dataset class which stores sparse contact matrix data in memory.

__init__(*data: numpy.ndarray*, *shape: Tuple[int, int, int]*, *scalars: Dict[str, numpy.ndarray] = {}*, *scalar_requires_grad: bool = False*)

Parameters

- **data** (*np.ndarray*) – Input contact matrices in sparse COO format of shape (N,) where N is the number of data examples, and the empty dimension is ragged. The row and column index vectors should be contatenated and the values are assumed to be 1 and don't need to be explcitly passed.
- **shape** (*Tuple[int, int, int]*) – Shape of the contact map (1, D, D) where D is the number of rows and columns.
- **scalars** (*Dict[str, np.ndarray], default={}*) – Dictionary of scalar arrays. For instance, the root mean squared deviation (RMSD) for each feature vector can be passed via `{"rmsd": np.array(...)}`. The dimension of each scalar array should match the number of input feature vectors N.
- **scalar_requires_grad** (*bool, default=False*) – Sets requires_grad torch.Tensor parameter for scalars specified by `scalars`. Set to True, to use scalars for multi-task learning. If scalars are only required for plotting, then set it as False.

class mdlearn.data.datasets.contact_map.**ContactMapHDF5Dataset**(**args: Any*, ***kwargs: Any*)

PyTorch Dataset class to load contact matrix data from HDF5 format.

__init__(*path: Union[str, pathlib.Path]*, *shape: Tuple[int, ...]*, *dataset_name: str = 'contact_map'*, *scalar_dset_names: List[str] = []*, *values_dset_name: Optional[str] = None*, *scalar_requires_grad: bool = False*, *in_memory: bool = True*)

Parameters

- **path** (*PathLike*) – Path to HDF5 file containing contact matrices.
- **shape** (*Tuple[int, ...]*) – Shape of contact matrices required by the model (H, W), may be (1, H, W).
- **dataset_name** (*str*) – Name of contact map dataset in HDF5 file.
- **scalar_dset_names** (*List[str]*) – List of scalar dataset names inside HDF5 file to be passed to training logs.
- **values_dset_name** (*str, optional*) – Name of HDF5 dataset field containing optional values of the entries the distance/contact matrix. By default, values are all assumed to be 1 corresponding to a binary contact map and created on the fly.
- **scalar_requires_grad** (*bool*) – Sets requires_grad torch.Tensor parameter for scalars specified by *scalar_dset_names*. Set to True, to use scalars for multi-task learning. If scalars are only required for plotting, then set it as False.
- **in_memory** (*bool*) – If True, pull data stored in HDF5 from disk to numpy arrays. Otherwise, read each batch from HDF5 on the fly.

Examples

```
>>> dataset = ContactMapDataset("contact_maps.h5", (28, 28))
>>> dataset[0]
{'X': torch.Tensor(..., dtype=float32), 'index': tensor(0)}
>>> dataset[0]["X"].shape
(28, 28)
```

```
>>> dataset = ContactMapDataset("contact_maps.h5", (28, 28), scalar_dset_names=[
↪"rmsd"])
>>> dataset[0]
{'X': torch.Tensor(..., dtype=float32), 'index': tensor(0), 'rmsd': tensor(8.
↪7578, dtype=torch.float16)}
```

mdlearn.data.datasets.feature_vector

Classes

FeatureVectorDataset(*args, **kwargs)	PyTorch Dataset class to load vector or scalar data directly from a np.ndarray.
FeatureVectorHDF5Dataset(*args, **kwargs)	PyTorch Dataset class to load vector or scalar data from an HDF5 file.
TimeFeatureVectorDataset(*args, **kwargs)	PyTorch Dataset class to handle time series feature vectors and optional scalars directly from a np.ndarray.

class mdlearn.data.datasets.feature_vector.**FeatureVectorDataset**(**args: Any*, ***kwargs: Any*)

PyTorch Dataset class to load vector or scalar data directly from a np.ndarray.

__init__(*data: numpy.ndarray*, *scalars: Dict[str, numpy.ndarray] = {}*, *scalar_requires_grad: bool = False*, *in_gpu_memory: bool = False*)

Parameters

- **data** (*np.ndarray*) – Input features vectors of shape (N, D) where N is the number of data examples, and D is the dimension of the feature vector.
- **scalars** (*Dict[str, np.ndarray], default={}*) – Dictionary of scalar arrays. For instance, the root mean squared deviation (RMSD) for each feature vector can be passed via `{"rmsd": np.array(...)}`. The dimension of each scalar array should match the number of input feature vectors N.
- **scalar_requires_grad** (*bool, default=False*) – Sets requires_grad torch.Tensor parameter for scalars specified by `scalars`. Set to True, to use scalars for multi-task learning. If scalars are only required for plotting, then set it as False.
- **in_gpu_memory** (*bool, default=False*) – If True, will pre-load the entire `data` array to GPU memory.

class mdlearn.data.datasets.feature_vector.**FeatureVectorHDF5Dataset**(**args: Any*, ***kwargs: Any*)

PyTorch Dataset class to load vector or scalar data from an HDF5 file.

__init__(*path: Union[str, pathlib.Path]*, *dataset_name: str*, *scalar_dset_names: List[str] = []*, *scalar_requires_grad: bool = False*, *in_memory: bool = True*)

Parameters

- **path** (*PathLike*) – Path to h5 file containing contact matrices.
- **dataset_name** (*str*) – Path to contact maps in HDF5 file.
- **scalar_dset_names** (*List[str], default=[]*) – List of scalar dataset names inside HDF5 file to be passed to training logs.
- **scalar_requires_grad** (*bool, default=False*) – Sets requires_grad torch.Tensor parameter for scalars specified by *scalar_dset_names*. Set to True, to use scalars for multi-task learning. If scalars are only required for plotting, then set it as False.
- **in_memory** (*bool, default=True*) – If True, pull data stored in HDF5 from disk to numpy arrays. Otherwise, read each batch from HDF5 on the fly.

class mdlearn.data.datasets.feature_vector.**TimeFeatureVectorDataset**(**args: Any*, ***kwargs: Any*)

PyTorch Dataset class to handle time series feature vectors and optional scalars directly from a np.ndarray.

__init__(*data: numpy.ndarray*, *scalars: Dict[str, numpy.ndarray] = {}*, *scalar_requires_grad: bool = False*, *in_gpu_memory: bool = False*, *window_size: int = 10*, *horizon: int = 1*)

Parameters

- **window_size** (*int, default=10*) – Number of timesteps considered for prediction.
- **horizon** (*int, default=1*) – How many time steps to predict ahead.

Raises `ValueError` – If the sum of `window_size` and `horizon` is longer than the input data.

mdlearn.data.datasets.point_cloud

PointCloud Dataset.

Classes

CenterOfMassTransform(data)	

PointCloudDataset(*args, **kwargs)	PyTorch Dataset class to load point cloud data.
PointCloudDatasetInMemory(*args, **kwargs)	PyTorch Dataset class to load point cloud data.

class mdlearn.data.datasets.point_cloud.**CenterOfMassTransform**(*data: numpy.ndarray*)

__init__(*data: numpy.ndarray*) → None

Computes center of mass transformation

Parameters **data** (*np.ndarray*) – Dataset of positions with shape (num_examples, 3, num_points).

transform(*x: numpy.ndarray*) → numpy.ndarray

Normalize example by bias and scale factors

Parameters **x** (*np.ndarray*) – Data to transform shape (3, num_points). Modifies `x`.

Returns *np.ndarray* – The transformed data

Raises **ValueError** – If NaN encountered in input

class mdlearn.data.datasets.point_cloud.**PointCloudDataset**(**args: Any*, ***kwargs: Any*)

PyTorch Dataset class to load point cloud data. Optionally, uses HDF5 files to only read into memory what is necessary for one batch.

__init__(*path: Union[str, pathlib.Path]*, *num_points: int*, *num_features: int = 0*, *dataset_name: str = 'point_cloud'*, *scalar_dset_names: List[str] = []*, *seed: int = 333*, *cms_transform: bool = False*, *scalar_requires_grad: bool = False*, *in_memory: bool = True*)

Parameters

- **path** (*Union[str, Path]*) – Path to HDF5 file containing data set.
- **dataset_name** (*str*) – Name of the point cloud data in the HDF5 file.
- **scalar_dset_names** (*List[str]*) – List of scalar dataset names inside HDF5 file to be passed to training logs.
- **num_points** (*int*) – Number of points per sample. Should be smaller or equal than the total number of points.
- **num_features** (*int*) – Number of additional per-point features in addition to xyz coords.
- **seed** (*int*) – Seed for the RNG for the splitting. Make sure it is the same for all workers reading from the same file.
- **cms_transform** (*bool*) – If True, subtract center of mass from batch and shift and scale batch by the full dataset statistics.
- **scalar_requires_grad** (*bool*) – Sets requires_grad torch.Tensor parameter for scalars specified by `scalar_dset_names`. Set to True, to use scalars for multi-task learning. If scalars are only required for plotting, then set it as False.

- **in_memory** (*bool*) – If True, pull data stored in HDF5 from disk to numpy arrays. Otherwise, read each batch from HDF5 on the fly.

Examples

```
>>> dataset = PointCloudDataset("point_clouds.h5", 28)
>>> dataset[0]
{'X': torch.Tensor(..., dtype=float32), 'index': tensor(0)}
>>> dataset[0]["X"].shape
torch.Size([3, 28])
```

```
>>> dataset = PointCloudDataset("point_clouds.h5", 28, 1)
>>> dataset[0]["X"].shape
torch.Size([4, 28])
```

```
>>> dataset = PointCloudDataset("point_clouds.h5", 28, scalar_dset_names=["rmsd
↪"])
>>> dataset[0]
{'X': torch.Tensor(..., dtype=float32), 'index': tensor(0), 'rmsd': tensor(8.
↪7578, dtype=torch.float16)}
```

property point_cloud_size: Tuple[int, int]

class mdlearn.data.datasets.point_cloud.**PointCloudDatasetInMemory**(**args: Any*, ***kwargs: Any*)

PyTorch Dataset class to load point cloud data. Optionally, uses HDF5 files to only read into memory what is necessary for one batch.

__init__(*data: numpy.ndarray*, *scalars: Dict[str, numpy.ndarray] = {}*, *cms_transform: bool = False*, *scalar_requires_grad: bool = False*)

Parameters

- **data** (*np.ndarray*) – Dataset of positions with shape (num_examples, 3, num_points)
- **scalars** (*Dict[str, np.ndarray], default={}*) – Dictionary of scalar arrays. For instance, the root mean squared deviation (RMSD) for each feature vector can be passed via `{"rmsd": np.array(...)}`. The dimension of each scalar array should match the number of input feature vectors N.
- **cms_transform** (*bool*) – If True, subtract center of mass from batch and shift and scale batch by the full dataset statistics.
- **scalar_requires_grad** (*bool*) – Sets requires_grad torch.Tensor parameter for scalars specified by `scalar_dset_names`. Set to True, to use scalars for learning. If scalars are only required for plotting, then set it as False.

mdlearn.data.datasets.time_contact_map

ContactMapTimeSeriesDataset Dataset.

Classes

ContactMapTimeSeriesDataset(*args, **kwargs)	PyTorch Dataset class to load contact matrix data in the format of a time series.

class mdlearn.data.datasets.time_contact_map.**ContactMapTimeSeriesDataset**(**args: Any*, ***kwargs: Any*)

PyTorch Dataset class to load contact matrix data in the format of a time series.

__init__(*path: Union[str, pathlib.Path]*, *shape: Tuple[int, ...]*, *lag_time: int = 1*, *dataset_name: str = 'contact_map'*, *scalar_dset_names: List[str] = []*, *values_dset_name: Optional[str] = None*, *scalar_requires_grad: bool = False*, *in_memory: bool = True*)

Parameters

- **path** (*PathLike*) – Path to HDF5 file containing contact matrices.
- **shape** (*Tuple[int, ...]*) – Shape of contact matrices required by the model (H, W), may be (1, H, W).
- **lag_time** (*int*) – Delay time forward or backward in the input data. The time-lagged correlations is computed between `X[t]` and `X[t+lag_time]`.
- **dataset_name** (*str*) – Name of contact map dataset in HDF5 file.
- **scalar_dset_names** (*List[str]*) – List of scalar dataset names inside HDF5 file to be passed to training logs.
- **values_dset_name** (*str, optional*) – Name of HDF5 dataset field containing optional values of the entries the distance/contact matrix. By default, values are all assumed to be 1 corresponding to a binary contact map and created on the fly.
- **scalar_requires_grad** (*bool*) – Sets requires_grad torch.Tensor parameter for scalars specified by *scalar_dset_names*. Set to True, to use scalars for multi-task learning. If scalars are only required for plotting, then set it as False.
- **in_memory** (*bool*) – If True, pull data stored in HDF5 from disk to numpy arrays. Otherwise, read each batch from HDF5 on the fly.

Examples

```
>>> dataset = ContactMapTimeSeriesDataset("contact_maps.h5", (28, 28))
>>> dataset[0]
{'X_t': torch.Tensor(..., dtype=float32), 'X_t_tau': torch.Tensor(...,␣
↪dtype=float32), 'index': tensor(0)}
>>> dataset[0]["X_t"].shape
(28, 28)
>>> dataset[0]["X_t_tau"].shape
(28, 28)
```

mdlearn.data.preprocess

Modules

mdlearn.data.preprocess.align
mdlearn.data.preprocess.decorrelation

mdlearn.data.preprocess.align

Modules

mdlearn.data.preprocess.align.iterative_means_align(coords)	Run iterative means alignment.
mdlearn.data.preprocess.align.kabsch_align	

mdlearn.data.preprocess.align.iterative_means_align

mdlearn.data.preprocess.align.**iterative_means_align**(*coords: numpy.ndarray*, *eps: float = 0.001*, *max_iter: int = 10*, *inplace: bool = False*, *verbose: bool = False*, *num_workers: int = 1*) → Tuple[int, List[numpy.ndarray], List[numpy.ndarray], numpy.ndarray]

Run iterative means alignment.

Run iterative means alignment which aligns `coords` to the mean coordinate structure using the kabsch alignment algorithm implemented here: *mdlearn.data.preprocess.align.kabsch_align*. Algorithm converges if either the difference of means coordinates computed from consecutive iterations is less than `eps` or if `max_iter` iterations have finished.

Parameters

- **coords** (*np.ndarray*) – Array of atomic coordinates with dimension (n_frames, 3, n_atoms)
- **eps** (*float, default=0.001*) – Error tolerance of the difference between mean coordinates computed from consecutive iterations, used to define convergence.
- **max_iter** (*int, default=10*) – Number of iterations before convergence.
- **inplace** (*bool, default=False*) – If True, modifies `coords` inplace. Inplace operations may offer the ability to fit larger systems into memory.
- **verbose** (*bool, default=False*) – If True, prints verbose output
- **num_workers** (*int, 1*) – Number of workers for parallel processing of the trajectory, each worker will take a single core.

Returns

- **itr** (*int*) – The iteration reached before convergence.
- **avg_coords** (*List[np.ndarray]*) – The average taken over all coords each iteration of the alignment.

- **e_rmsd** (*List[np.ndarray]*) – The root mean squared deviation (RMSD) of each structure with respect to the `avg_coords` for each iteration of the alignment.
- **coords_** (*np.ndarray*) – The newly aligned trajectory of coordinates with the same shape as the input `coords` array.

mdlearn.data.preprocess.align.kabsch_align

Functions

kabsch(to_xyz, from_xyz[, return_err])	Aligns a single frame `from_xyz` to another frame `to_xyz` using the kabsch method.

mdlearn.data.preprocess.align.kabsch_align.**kabsch**(*to_xyz: numpy.ndarray*, *from_xyz: numpy.ndarray*, *return_err: bool = True*) → Tuple[float, numpy.ndarray, Optional[numpy.ndarray]]

Aligns a single frame `from_xyz` to another frame `to_xyz` using the kabsch method.

Parameters

- **to_xyz** (*np.ndarray*) – 3 x N array of coordinates to align to.
- **from_xyz** (*np.ndarray*) – A 3 x N array of coordinates to align.
- **return_err** (*bool, default=True*) – Will return the errors.

Returns

- **e_rmsd** (*float*) – The root mean squared deviation (RMSD).
- **new_xyz** (*np.ndarray*) – The newly aligned coordinates with the same shape as `fromXYZ`.
- **err** (*Optional[np.ndarray]*) – Returns the raw error values if `return_err` is True, otherwise returns `None`.

Raises `ValueError` – If the arrays differ in the number of coordinates N.

mdlearn.data.preprocess.decorrelation

Modules

mdlearn.data.preprocess.decorrelation.spatial	Spatial decorrelation functions.
mdlearn.data.preprocess.decorrelation.temporal	

mdlearn.data.preprocess.decorrelation.spatial

Spatial decorrelation functions.

Functions

SD2(data[, m, verbose])	Perform spatial decorrelation of 2nd order of real signals.
SD4(Y[, m, U, verbose])	SD4 - Spatial Decorrelation of 4th order of real signals.

mdlearn.data.preprocess.decorrelation.spatial.**SD2**(*data: numpy.ndarray*, *m: Optional[int] = None*, *verbose: bool = False*)

Perform spatial decorrelation of 2nd order of real signals.

Parameters

- **data** (*np.ndarray*) – data array of shape (T, 3N) where T is the number of frames in the MD trajectory, N is the number of atoms in the system and 3 is due to the x,y,z coordinates for each atom.
- **m** (*Optional[int], default=None*) – Dimensionality of the subspace we are interested in. Default value is None, in which case m=n. If m is omitted, U is a square 3n x 3n matrix (as many sources as sensors).
- **verbose** (*bool, default=False*) – Print progress.

Returns

- **Y** (*np.ndarray*) – A 3n x m matrix U (NumPy matrix type), such that $Y = U \times$ `data` is a 2nd order spatially whitened source extracted from the 3n x T data matrix `data` by performing PCA on `m` components of the real data. `Y` is a matrix of spatially uncorrelated components.
- **S** (*np.ndarray*) – Eigen values of the `data` covariance matrix.
- **B** (*np.ndarray*) – Eigen vectors of the `data` covariance matrix. The eigen vectors are orthogonal.
- **U** (*np.ndarray*) – The sphering matrix used to transform `data` by $Y = U \times$ `data`.

Raises

- **TypeError** – If `verbose` is not of type bool.
- **TypeError** – If `data` is not of type np.ndarray.
- **ValueError** – If `data` does not have 2 dimensions.
- **ValueError** – If `m` is greater than 3N, the second dimension of `data`.

mdlearn.data.preprocess.decorrelation.spatial.**SD4**(*Y: numpy.ndarray*, *m: Optional[int] = None*, *U: Optional[numpy.ndarray] = None*, *verbose: bool = False*) → numpy.ndarray

SD4 - Spatial Decorrelation of 4th order of real signals.

SD4 does joint diagonalization of cumulant matrices of order 4 to decorrelate the signals in spatial domain. It allows us to extract signals which are as independent as possible and which were not obtained while performing SD2. Here we consider signals which are spatially decorrelated of order 2, meaning that SD2 should be run first.

Parameters

- **Y** (*np.ndarray*) – An `n x T` spatially whitened matrix (`n` subspaces, `T` samples). May be a numpy array or matrix where `n` is the number of subspaces we are interested in and `T` is the number of frames in the MD trajectory.
- **m** (*Optional[int], default=None*) – The number of subspaces we are interested in. Defaults to None, in which case m=k.
- **U** (*Optional[np.ndarray], default=None*) – Whitening matrix obtained after doing the PCA analysis on `n` components of real data.
- **verbose** (*bool, default=False*) – Print progress.

Returns **W** (*np.ndarray*) – Separating matrix which is spatially decorrelated of 4th order.

Raises `ValueError` – If `m` is greater than `n`, the first dimension of `Y`.

mdlearn.data.preprocess.decorrelation.temporal

mdlearn.data.utils

Utility functions for handling PyTorch data objects.

Functions

train_valid_split(dataset[, split_pct, method])	Creates training and validation DataLoaders from `dataset`.

mdlearn.data.utils.**train_valid_split**(*dataset: torch.utils.data.Dataset*, *split_pct: float = 0.8*, *method: str = 'random'*, ***kwargs*) → Tuple[torch.utils.data.DataLoader, torch.utils.data.DataLoader]

Creates training and validation DataLoaders from `dataset`.

Parameters

- **dataset** (*Dataset*) – A PyTorch dataset class derived from `torch.utils.data.Dataset`.
- **split_pct** (*float*) – Percentage of data to be used as training data after a split.
- **method** (*str, default="random"*) – Method to split the data. For random split use "random", for a simple partition, use "partition".
- ****kwargs** – Keyword arguments to `torch.utils.data.DataLoader`. Includes, `batch_size`, `drop_last`, etc (see PyTorch Docs).

Raises `ValueError` – If `method` is not "random" or "partition".

mdlearn.metrics

Functions

metric_cluster_quality(data, metric[, ...])	

mdlearn.metrics.**metric_cluster_quality**(*data: numpy.ndarray*, *metric: numpy.ndarray*, *n_samples: int = 10000*, *n_neighbors: int = 10*) → float

mdlearn.nn

Modules

mdlearn.nn.models	
mdlearn.nn.modules	
mdlearn.nn.utils	

mdlearn.nn.models

Modules

mdlearn.nn.models.aae	
mdlearn.nn.models.ae	
mdlearn.nn.models.lstm	**Warning:** LSTM models are still under development, use with caution!
mdlearn.nn.models.vae	
mdlearn.nn.models.vde	
mdlearn.nn.models.wae	

mdlearn.nn.models.aae

Modules

mdlearn.nn.models.aae.model	
mdlearn.nn.models.aae.point_3d_aae	Adversarial Autoencoder for 3D point cloud data (3dAAE)

mdlearn.nn.models.aae.model

Classes

AAE(*args, **kwargs)	
ChamferLoss(*args, **kwargs)	

class mdlearn.nn.models.aae.model.**AAE**(**args: Any*, ***kwargs: Any*)

discriminate(**args*, ***kwargs*) → torch.Tensor

Discriminator forward pass.

Parameters

- ***args** – Variable length discriminator argument list.
- ****kwargs** – Arbitrary discriminator keyword arguments.

Returns *torch.Tensor* – The discriminator output.

reset_parameters() → None

Reset encoder, decoder and discriminator parameters.

class mdlearn.nn.models.aae.model.**ChamferLoss**(**args: Any*, ***kwargs: Any*)

batch_pairwise_dist(*x: torch.Tensor*, *y: torch.Tensor*) → torch.Tensor

forward(*preds: torch.Tensor*, *gts: torch.Tensor*) → torch.Tensor

mdlearn.nn.models.aae.point_3d_aae

Adversarial Autoencoder for 3D point cloud data (3dAAE)

Classes

AAE3d(*args, **kwargs)	

class mdlearn.nn.models.aae.point_3d_aae.**AAE3d**(**args: Any*, ***kwargs: Any*)

__init__(*num_points: int*, *num_features: int = 0*, *latent_dim: int = 20*, *encoder_bias: bool = True*, *encoder_relu_slope: float = 0.0*, *encoder_filters: List[int] = [64, 128, 256, 256, 512]*, *encoder_kernels: List[int] = [5, 5, 3, 1, 1]*, *decoder_bias: bool = True*, *decoder_relu_slope: float = 0.0*, *decoder_affine_widths: List[int] = [64, 128, 512, 1024]*, *discriminator_bias: bool = True*, *discriminator_relu_slope: float = 0.0*, *discriminator_affine_widths: List[int] = [512, 128, 64]*)

Adversarial Autoencoder module for point cloud data from the "Adversarial Autoencoders for Compact Representations of 3D Point Clouds" paper and adapted to work on atomic coordinate data in the "AI-Driven Multiscale Simulations Illuminate Mechanisms of SARS-CoV-2 Spike Dynamics" paper. Inherits from mdlearn.nn.models.aae.AAE.

Parameters

- **num_points** (*int*) – Number of input points in point cloud.
- **num_features** (*int, optional*) – Number of scalar features per point in addition to 3D coordinates, by default 0
- **latent_dim** (*int, optional*) – Latent dimension of the encoder, by default 20
- **encoder_bias** (*bool, optional*) – Use a bias term in the encoder Conv1d layers, by default True.
- **encoder_relu_slope** (*float, optional*) – If greater than 0.0, will use LeakyReLU activiation in the encoder with `negative_slope` set to `relu_slope`, by default 0.0
- **encoder_filters** (*List[int], optional*) – Encoder Conv1d filter sizes, by default [64, 128, 256, 256, 512]
- **encoder_kernels** (*List[int], optional*) – Encoder Conv1d kernel sizes, by default [5, 5, 3, 1, 1]
- **decoder_bias** (*bool, optional*) – Use a bias term in the decoder Linear layers, by default True
- **decoder_relu_slope** (*float, optional*) – If greater than 0.0, will use LeakyReLU activiation in the decoder with `negative_slope` set to `relu_slope`, by default 0.0
- **decoder_affine_widths** (*List[int], optional*) – Decoder Linear layers `in_features`, by default [64, 128, 512, 1024]
- **discriminator_bias** (*bool, optional*) – Use a bias term in the discriminator Linear layers, by default True.
- **discriminator_relu_slope** (*float, optional*) – If greater than 0.0, will use LeakyReLU activiation in the discriminator with `negative_slope` set to `relu_slope`, by default 0.0
- **discriminator_affine_widths** (*List[int], optional*) – Discriminator Linear layers `in_features`, by default [512, 128, 64]

`critic_loss`(*real_logits: torch.Tensor*, *fake_logits: torch.Tensor*) → torch.Tensor

Classification loss (critic) function.

Parameters

- **real_logits** (*torch.Tensor*) – Discriminator output logits from prior distribution.
- **fake_logits** (*torch.Tensor*) – Discriminator output logits from encoded latent vectors.

Returns *torch.Tensor* – Classification loss i.e. the difference between logit means.

`decoder_loss`(*fake_logit: torch.Tensor*) → torch.Tensor

Decoder/Generator loss.

Parameters **fake_logit** (*torch.Tensor*) – Output of discriminator.

Returns *torch.Tensor* – Negative mean of the fake logits.

`forward`(*x: torch.Tensor*) → Tuple[torch.Tensor, torch.Tensor]

Forward pass of encoder and decoder.

Parameters **x** (*torch.Tensor*) – Input point cloud data.

Returns *Tuple[torch.Tensor, torch.Tensor]* – The z-latent vector, and the `recon_x` reconstruction.

gp_loss(*noise: torch.Tensor*, *z: torch.Tensor*) → torch.Tensor

Gradient penalty loss function.

Parameters

- **noise** (*[type]*) – Random noise sampled from prior distribution.
- **z** (*[type]*) – Encoded latent vectors.

Returns *torch.Tensor* – The gradient penalty loss.

recon_loss(*x: torch.Tensor*, *recon_x: torch.Tensor*) → torch.Tensor

Reconstruction loss using ChamferLoss.

Parameters

- **x** (*torch.Tensor*) – The original input tensor.
- **recon_x** (*torch.Tensor*) – The reconstructed output tensor.

Returns *torch.Tensor* – Reconstruction loss measured by Chamfer distance.

mdlearn.nn.models.ae

Modules

mdlearn.nn.models.ae.linear	Linear-layer autoencoder model with trainer class.
mdlearn.nn.models.ae.lstm	

Warning: LSTM models are still under development, use with caution!

mdlearn.nn.models.ae.model	

mdlearn.nn.models.ae.linear

Linear-layer autoencoder model with trainer class.

Classes

LinearAE(*args, **kwargs)	A symmetric autoencoder with all linear layers.
LinearAETrainer([input_dim, latent_dim, ...])	Trainer class to fit a linear autoencoder to a set of feature vectors.

class mdlearn.nn.models.ae.linear.**LinearAE**(**args: Any*, ***kwargs: Any*)

A symmetric autoencoder with all linear layers. Applies a ReLU activation between encoder and decoder.

__init__(*input_dim: int*, *latent_dim: int = 8*, *hidden_neurons: List[int] = [128]*, *bias: bool = True*, *relu_slope: float = 0.0*, *inplace_activation: bool = False*)

Parameters

- **input_dim** (*int*) – Dimension of input tensor (should be flattened).
- **latent_dim** (*int, default=8*) – Dimension of the latent space.
- **hidden_neurons** (*List[int], default=[128]*) – Linear layers `in_features`.
- **bias** (*bool, default=True*) – Use a bias term in the Linear layers.
- **relu_slope** (*float, default=0.0*) – If greater than 0.0, will use LeakyReLU activiation with `negative_slope` set to `relu_slope`.
- **inplace_activation** (*bool, default=False*) – Sets the inplace option for the activation function.

forward(*x: torch.Tensor*) → Tuple[torch.Tensor, torch.Tensor]

Forward pass of autoencoder.

Parameters **x** (*torch.Tensor*) – Input data.

Returns *Tuple[torch.Tensor, torch.Tensor]* – The batch of latent vectors `z` and the reconstructions `recon_x`.

recon_loss(*x: torch.Tensor*, *recon_x: torch.Tensor*, *reduction: str = 'mean'*) → torch.Tensor

Compute the MSE reconstruction loss between `x` and `recon_x`.

Parameters

- **x** (*torch.Tensor*) – The input data.
- **recon_x** (*torch.Tensor*) – The reconstruction of the input data `x`
- **reduction** (*str, default="mean"*) – The reduction strategy for the F.mse_loss function.

Returns *torch.Tensor* – The reconstruction loss between `x` and `recon_x`.

class mdlearn.nn.models.ae.linear.**LinearAETrainer**(*input_dim: int = 40*, *latent_dim: int = 3*, *hidden_neurons: List[int] = [32, 16, 8]*, *bias: bool = True*, *relu_slope: float = 0.0*, *inplace_activation: bool = False*, *seed: int = 42*, *in_gpu_memory: bool = False*, *num_data_workers: int = 0*, *prefetch_factor: int = 2*, *split_pct: float = 0.8*, *split_method: str = 'random'*, *batch_size: int = 128*, *shuffle: bool = True*, *device: str = 'cpu'*, *optimizer_name: str = 'RMSprop'*, *optimizer_hparams: Dict[str, Any] = {'lr': 0.001, 'weight_decay': 1e-05}*, *scheduler_name: Optional[str] = None*, *scheduler_hparams: Dict[str, Any] = {}*, *epochs: int = 100*, *verbose: bool = False*, *clip_grad_max_norm: float = 10.0*, *checkpoint_log_every: int = 10*, *plot_log_every: int = 10*, *plot_n_samples: int = 10000*, *plot_method: Optional[str] = 'TSNE'*, *train_subsample_pct: float = 1.0*, *valid_subsample_pct: float = 1.0*, *use_wandb: bool = False*)

Trainer class to fit a linear autoencoder to a set of feature vectors.

__init__(*input_dim: int = 40*, *latent_dim: int = 3*, *hidden_neurons: List[int] = [32, 16, 8]*, *bias: bool = True*, *relu_slope: float = 0.0*, *inplace_activation: bool = False*, *seed: int = 42*, *in_gpu_memory: bool = False*, *num_data_workers: int = 0*, *prefetch_factor: int = 2*, *split_pct: float = 0.8*, *split_method: str = 'random'*, *batch_size: int = 128*, *shuffle: bool = True*, *device: str = 'cpu'*, *optimizer_name: str = 'RMSprop'*, *optimizer_hparams: Dict[str, Any] = {'lr': 0.001, 'weight_decay': 1e-05}*, *scheduler_name: Optional[str] = None*, *scheduler_hparams: Dict[str, Any] = {}*, *epochs: int = 100*, *verbose: bool = False*, *clip_grad_max_norm: float = 10.0*, *checkpoint_log_every: int = 10*, *plot_log_every: int = 10*, *plot_n_samples: int = 10000*, *plot_method: Optional[str] = 'TSNE'*, *train_subsample_pct: float = 1.0*, *valid_subsample_pct: float = 1.0*, *use_wandb: bool = False*)

Parameters

- **input_dim** (*int, default=40*) – Dimension of input tensor (should be flattened).
- **latent_dim** (*int, default=3*) – Dimension of the latent space.
- **hidden_neurons** (*List[int], default=[32, 16, 8]*) – Linear layers `in_features`. Defines the shape of the autoencoder (does not include latent dimension). The encoder and decoder are symmetric.
- **bias** (*bool, default=True*) – Use a bias term in the Linear layers.
- **relu_slope** (*float, default=0.0*) – If greater than 0.0, will use LeakyReLU activiation with `negative_slope` set to `relu_slope`.
- **inplace_activation** (*bool, default=False*) – Sets the inplace option for the activation function.
- **seed** (*int, default=42*) – Random seed for torch, numpy, and random module.
- **in_gpu_memory** (*bool, default=False*) – If True, will pre-load the entire `data` array to GPU memory.
- **num_data_workers** (*int, default=0*) – How many subprocesses to use for data loading. 0 means that the data will be loaded in the main process.

- **prefetch_factor** (*int, by default=2*) – Number of samples loaded in advance by each worker. 2 means there will be a total of 2 * num_workers samples prefetched across all workers.
- **split_pct** (*float, default=0.8*) – Proportion of data set to use for training. The rest goes to validation.
- **split_method** (*str, default="random"*) – Method to split the data. For random split use "random", for a simple partition, use "partition".
- **batch_size** (*int, default=128*) – Mini-batch size for training.
- **shuffle** (*bool, default=True*) – Whether to shuffle training data or not.
- **device** (*str, default="cpu"*) – Specify training hardware either `cpu` or `cuda` for GPU devices.
- **optimizer_name** (*str, default="RMSprop"*) – Name of the PyTorch optimizer to use. Matches PyTorch optimizer class name.
- **optimizer_hparams** (*Dict[str, Any], default={"lr": 0.001, "weight_decay": 0.00001}*) – Dictionary of hyperparameters to pass to the chosen PyTorch optimizer.
- **scheduler_name** (*Optional[str], default=None*) – Name of the PyTorch learning rate scheduler to use. Matches PyTorch optimizer class name.
- **scheduler_hparams** (*Dict[str, Any], default={}*) – Dictionary of hyperparameters to pass to the chosen PyTorch learning rate scheduler.
- **epochs** (*int, default=100*) – Number of epochs to train for.
- **verbose** (*bool, default False*) – If True, will print training and validation loss at each epoch.
- **clip_grad_max_norm** (*float, default=10.0*) – Max norm of the gradients for gradient clipping for more information see: `torch.nn.utils.clip_grad_norm_` documentation.
- **checkpoint_log_every** (*int, default=10*) – Epoch interval to log a checkpoint file containing the model weights, optimizer, and scheduler parameters.
- **plot_log_every** (*int, default=10*) – Epoch interval to log a visualization plot of the latent space.
- **plot_n_samples** (*int, default=10000*) – Number of validation samples to use for plotting.
- **plot_method** (*Optional[str], default="TSNE"*) – The method for visualizing the latent space or if visualization should not be run, set `plot_method=None`. If using `"TSNE"`, it will attempt to use the RAPIDS.ai GPU implementation and will fallback to the sklearn CPU implementation if RAPIDS.ai is unavailable.
- **train_subsample_pct** (*float, default=1.0*) – Percentage of training data to use during hyperparameter sweeps.
- **valid_subsample_pct** (*float, default=1.0*) – Percentage of validation data to use during hyperparameter sweeps.
- **use_wandb** (*bool, default=False*) – If True, will log results to wandb.

Raises

- **ValueError** – `split_pct` should be between 0 and 1.
- **ValueError** – `train_subsample_pct` should be between 0 and 1.
- **ValueError** – `valid_subsample_pct` should be between 0 and 1.
- **ValueError** – Specified `device` as `cuda`, but it is unavailable.

`fit`(*X: numpy.ndarray*, *scalars: Dict[str, numpy.ndarray] = {}*, *output_path: Union[str, pathlib.Path] = './'*, *checkpoint: Optional[Union[str, pathlib.Path]] = None*)

Trains the autoencoder on the input data `X`.

Parameters

- **X** (*np.ndarray*) – Input features vectors of shape (N, D) where N is the number of data examples, and D is the dimension of the feature vector.
- **scalars** (*Dict[str, np.ndarray], default={}*) – Dictionary of scalar arrays. For instance, the root mean squared deviation (RMSD) for each feature vector can be passed via `{"rmsd": np.array(...)}`. The dimension of each scalar array should match the number of input feature vectors N.
- **output_path** (*PathLike, default="./"*) – Path to write training results to. Makes an `output_path/checkpoints` folder to save model checkpoint files, and `output_path/plots` folder to store latent space visualizations.
- **checkpoint** (*Optional[PathLike], default=None*) – Path to a specific model checkpoint file to restore training.

Raises

- `TypeError` – If `scalars` is not type dict. A common error is to pass `output_path` as the second argument.
- `NotImplementedError` – If using a learning rate scheduler other than `ReduceLROnPlateau`, a step function will need to be implemented.

`predict`(*X: numpy.ndarray*, *inference_batch_size: int = 512*, *checkpoint: Optional[Union[str, pathlib.Path]] = None*) → Tuple[numpy.ndarray, float]

Predict using the LinearAE

Parameters

- **X** (*np.ndarray*) – The input data to predict on.
- **inference_batch_size** (*int, default=512*) – The batch size for inference.
- **checkpoint** (*Optional[PathLike], default=None*) – Path to a specific model checkpoint file.

Returns *Tuple[np.ndarray, float]* – The `z` latent vectors corresponding to the input data `X` and the average reconstruction loss.

mdlearn.nn.models.ae.lstm

Warning: LSTM models are still under development, use with caution!

Classes

LSTMAE(*args, **kwargs)	LSTM model to predict the dynamics for a time series of feature vectors.
LSTMAETrainer(input_dim[, latent_dim, ...])	Trainer class to fit an LSTM model to a time series of feature vectors.

class mdlearn.nn.models.ae.lstm.**LSTMAE**(**args: Any*, ***kwargs: Any*)

LSTM model to predict the dynamics for a time series of feature vectors.

__init__(*input_dim: int*, *latent_dim: int = 8*, *hidden_neurons: List[int] = [128]*, *lstm_bias: bool = True*, *dropout: float = 0.0*, *relu_slope: float = 0.0*, *inplace_activation: bool = False*, *dense_bias: bool = True*)

Parameters

- **input_dim** (*int*) – The number of expected features in the input `x`.
- **latent_dim** (*int, default=8*) – Dimension of the latent space.
- **hidden_neurons** (*List[int], default=[128]*) – The dimension of the hidden states for each LSTM block in the stacked LSTM encoder. This list defines how deep the encoder is i.e. how many LSTM blocks to use. The reverse of this list also defines the shape of the `DenseNet` decoder.
- **lstm_bias** (*bool, default=True*) – If False, then the stacked LSTM encoder does not use bias weights b_ih and b_hh.
- **dropout** (*float, default=0.0*) – If non-zero, introduces a Dropout layer on the outputs of each LSTM layer except the last layer, with dropout probability equal to dropout.
- **relu_slope** (*float, default=0.0*) – If greater than 0.0, will use LeakyReLU activiation in the `DenseNet` decoder with `negative_slope` set to `relu_slope`.
- **inplace_activation** (*bool, default=False*) – Sets the inplace option for the activation function in the `DenseNet` decoder.
- **dense_bias** (*bool, default=True*) – If False, then the `DenseNet` decoder does not use bias.

forward(*x: torch.Tensor*) → Tuple[torch.Tensor, torch.Tensor]

Parameters **x** (*torch.Tensor*) – Tensor of shape BxNxD for B batches of length N sequences with D feature dimensions.

Returns

- *torch.Tensor* – The latent embedding of size (B, `latent_dim`).
- *torch.Tensor* – The predicted future time step of size (B, D).

mse_loss(*y_true: torch.Tensor*, *y_pred: torch.Tensor*, *reduction: str = 'mean'*) → torch.Tensor

Compute the MSE loss between `y_true` and `y_pred`.

Parameters

- **y_true** (*torch.Tensor*) – The true data.
- **y_pred** (*torch.Tensor*) – The prediction.
- **reduction** (*str, default="mean"*) – The reduction strategy for the F.mse_loss function.

Returns *torch.Tensor* – The MSE loss between `y_true` and `y_pred`.

class mdlearn.nn.models.ae.lstm.**LSTMAETrainer**(*input_dim: int*, *latent_dim: int = 8*, *hidden_neurons: List[int] = [128]*, *lstm_bias: bool = True*, *dropout: float = 0.0*, *relu_slope: float = 0.0*, *inplace_activation: bool = False*, *dense_bias: bool = True*, *window_size: int = 10*, *horizon: int = 1*, *seed: int = 42*, *in_gpu_memory: bool = False*, *num_data_workers: int = 0*, *prefetch_factor: int = 2*, *split_pct: float = 0.8*, *split_method: str = 'partition'*, *batch_size: int = 128*, *shuffle: bool = True*, *device: str = 'cpu'*, *optimizer_name: str = 'RMSprop'*, *optimizer_hparams: Dict[str, Any] = {'lr': 0.001, 'weight_decay': 1e-05}*, *scheduler_name: Optional[str] = None*, *scheduler_hparams: Dict[str, Any] = {}*, *epochs: int = 100*, *verbose: bool = False*, *clip_grad_max_norm: float = 10.0*, *checkpoint_log_every: int = 10*, *plot_log_every: int = 10*, *plot_n_samples: int = 10000*, *plot_method: Optional[str] = 'TSNE'*, *train_subsample_pct: float = 1.0*, *valid_subsample_pct: float = 1.0*, *use_wandb: bool = False*)

Trainer class to fit an LSTM model to a time series of feature vectors.

__init__(*input_dim: int*, *latent_dim: int = 8*, *hidden_neurons: List[int] = [128]*, *lstm_bias: bool = True*, *dropout: float = 0.0*, *relu_slope: float = 0.0*, *inplace_activation: bool = False*, *dense_bias: bool = True*, *window_size: int = 10*, *horizon: int = 1*, *seed: int = 42*, *in_gpu_memory: bool = False*, *num_data_workers: int = 0*, *prefetch_factor: int = 2*, *split_pct: float = 0.8*, *split_method: str = 'partition'*, *batch_size: int = 128*, *shuffle: bool = True*, *device: str = 'cpu'*, *optimizer_name: str = 'RMSprop'*, *optimizer_hparams: Dict[str, Any] = {'lr': 0.001, 'weight_decay': 1e-05}*, *scheduler_name: Optional[str] = None*, *scheduler_hparams: Dict[str, Any] = {}*, *epochs: int = 100*, *verbose: bool = False*, *clip_grad_max_norm: float = 10.0*, *checkpoint_log_every: int = 10*, *plot_log_every: int = 10*, *plot_n_samples: int = 10000*, *plot_method: Optional[str] = 'TSNE'*, *train_subsample_pct: float = 1.0*, *valid_subsample_pct: float = 1.0*, *use_wandb: bool = False*)

Parameters

- **input_dim** (*int*) – The number of expected features in the input `x`.
- **latent_dim** (*int, default=8*) – Dimension of the latent space.
- **hidden_neurons** (*List[int], default=[128]*) – The dimension of the hidden states for each LSTM block in the stacked LSTM encoder. This list defines how deep the encoder is i.e. how many LSTM blocks to use. The reverse of this list also defines the shape of the `DenseNet` decoder.
- **lstm_bias** (*bool, default=True*) – If False, then the stacked LSTM encoder does not use bias weights b_ih and b_hh.
- **dropout** (*float, default=0.0*) – If non-zero, introduces a Dropout layer on the outputs of each LSTM layer except the last layer, with dropout probability equal to dropout.
- **relu_slope** (*float, default=0.0*) – If greater than 0.0, will use LeakyReLU activiation in the `DenseNet` decoder with `negative_slope` set to `relu_slope`.
- **inplace_activation** (*bool, default=False*) – Sets the inplace option for the activation function in the `DenseNet` decoder.
- **dense_bias** (*bool, default=True*) – If False, then the `DenseNet` decoder does not use bias.
- **window_size** (*int, default=10*) – Number of timesteps considered for prediction.

- **horizon** (*int, default=1*) – How many time steps to predict ahead.
- **seed** (*int, default=42*) – Random seed for torch, numpy, and random module.
- **in_gpu_memory** (*bool, default=False*) – If True, will pre-load the entire `data` array to GPU memory.
- **num_data_workers** (*int, default=0*) – How many subprocesses to use for data loading. 0 means that the data will be loaded in the main process.
- **prefetch_factor** (*int, by default=2*) – Number of samples loaded in advance by each worker. 2 means there will be a total of 2 * num_workers samples prefetched across all workers.
- **split_pct** (*float, default=0.8*) – Proportion of data set to use for training. The rest goes to validation.
- **split_method** (*str, default="random"*) – Method to split the data. For random split use "random", for a simple partition, use "partition".
- **batch_size** (*int, default=128*) – Mini-batch size for training.
- **shuffle** (*bool, default=True*) – Whether to shuffle training data or not.
- **device** (*str, default="cpu"*) – Specify training hardware either `cpu` or `cuda` for GPU devices.
- **optimizer_name** (*str, default="RMSprop"*) – Name of the PyTorch optimizer to use. Matches PyTorch optimizer class name.
- **optimizer_hparams** (*Dict[str, Any], default={"lr": 0.001, "weight_decay": 0.00001}*) – Dictionary of hyperparameters to pass to the chosen PyTorch optimizer.
- **scheduler_name** (*Optional[str], default=None*) – Name of the PyTorch learning rate scheduler to use. Matches PyTorch optimizer class name.
- **scheduler_hparams** (*Dict[str, Any], default={}*) – Dictionary of hyperparameters to pass to the chosen PyTorch learning rate scheduler.
- **epochs** (*int, default=100*) – Number of epochs to train for.
- **verbose** (*bool, default=False*) – If True, will print training and validation loss at each epoch.
- **clip_grad_max_norm** (*float, default=10.0*) – Max norm of the gradients for gradient clipping for more information see: `torch.nn.utils.clip_grad_norm_` documentation.
- **checkpoint_log_every** (*int, default=10*) – Epoch interval to log a checkpoint file containing the model weights, optimizer, and scheduler parameters.
- **plot_log_every** (*int, default=10*) – Epoch interval to log a visualization plot of the latent space.
- **plot_n_samples** (*int, default=10000*) – Number of validation samples to use for plotting.
- **plot_method** (*Optional[str], default="TSNE"*) – The method for visualizing the latent space or if visualization should not be run, set `plot_method=None`. If using `"TSNE"`, it will attempt to use the RAPIDS.ai GPU implementation and will fallback to the sklearn CPU implementation if RAPIDS.ai is unavailable.
- **train_subsample_pct** (*float, default=1.0*) – Percentage of training data to use during hyperparameter sweeps.
- **valid_subsample_pct** (*float, default=1.0*) – Percentage of validation data to use during hyperparameter sweeps.

- **use_wandb** (*bool, default=False*) – If True, will log results to wandb.

Raises

- **ValueError** – `split_pct` should be between 0 and 1.
- **ValueError** – `train_subsample_pct` should be between 0 and 1.
- **ValueError** – `valid_subsample_pct` should be between 0 and 1.
- **ValueError** – Specified `device` as `cuda`, but it is unavailable.

fit(*X: numpy.ndarray*, *scalars: Dict[str, numpy.ndarray] = {}*, *output_path: Union[str, pathlib.Path] = './'*, *checkpoint: Optional[Union[str, pathlib.Path]] = None*)

Trains the LSTMAE on the input data `X`.

Parameters

- **X** (*np.ndarray*) – Input features vectors of shape (N, D) where N is the number of data examples, and D is the dimension of the feature vector.
- **scalars** (*Dict[str, np.ndarray], default={}*) – Dictionary of scalar arrays. For instance, the root mean squared deviation (RMSD) for each feature vector can be passed via `{"rmsd": np.array(...)}`. The dimension of each scalar array should match the number of input feature vectors N.
- **output_path** (*PathLike, default="./"*) – Path to write training results to. Makes an `output_path/checkpoints` folder to save model checkpoint files, and `output_path/plots` folder to store latent space visualizations.
- **checkpoint** (*Optional[PathLike], default=None*) – Path to a specific model checkpoint file to restore training.

Raises

- **ValueError** – If `X` does not have two dimensions. For scalar time series, please reshape to (N, 1).
- **TypeError** – If `scalars` is not type dict. A common error is to pass `output_path` as the second argument.
- **NotImplementedError** – If using a learning rate scheduler other than `ReduceLROnPlateau`, a step function will need to be implemented.

predict(*X: numpy.ndarray*, *inference_batch_size: int = 512*, *checkpoint: Optional[Union[str, pathlib.Path]] = None*) → Tuple[numpy.ndarray, numpy.ndarray, float]

Predict using the LSTMAE.

Parameters

- **X** (*np.ndarray*) – The input data to predict on.
- **inference_batch_size** (*int, default=512*) – The batch size for inference.
- **checkpoint** (*Optional[PathLike], default=None*) – Path to a specific model checkpoint file.

Returns

- *np.ndarray* – The predictions.
- *np.ndarray* – The latenet embeddings.
- *float* – The average MSE loss.

mdlearn.nn.models.ae.model

Classes

AE(*args, **kwargs)	Autoencoder base class module.

class mdlearn.nn.models.ae.model.**AE**(**args: Any*, ***kwargs: Any*)

Autoencoder base class module.

__init__(*encoder: torch.nn.Module*, *decoder: torch.nn.Module*)

Parameters

- **encoder** (*torch.nn.Module*) – The encoder module.
- **decoder** (*torch.nn.Module*) – The decoder module.

decode(**args*, ***kwargs*)

Decoder forward pass.

encode(**args*, ***kwargs*)

Encoder forward pass.

recon_loss(*x: torch.Tensor*, *recon_x: torch.Tensor*) → torch.Tensor

Compute the reconstruction loss between `x` and `recon_x`.

Parameters

- **x** (*torch.Tensor*) – The input data.
- **recon_x** (*torch.Tensor*) – The reconstruction of the input data `x`.

Returns *torch.Tensor* – The reconstruction loss between `x` and `recon_x`.

reset_parameters()

Reset encoder and decoder parameters.

mdlearn.nn.models.lstm

Warning: LSTM models are still under development, use with caution!

Classes

LSTM(*args, **kwargs)	LSTM model to predict the dynamics for a time series of feature vectors.
LSTMTrainer(input_size[, hidden_size, ...])	Trainer class to fit an LSTM model to a time series of feature vectors.

class mdlearn.nn.models.lstm.**LSTM**(**args: Any*, ***kwargs: Any*)

LSTM model to predict the dynamics for a time series of feature vectors.

`__init__`(*input_size: int*, *hidden_size: Optional[int] = None*, *num_layers: int = 1*, *bias: bool = True*, *dropout: float = 0.0*, *bidirectional: bool = False*)

Parameters

- **input_size** (*int*) – The number of expected features in the input `x`.
- **hidden_size** (*Optional[int], default=None*) – The number of features in the hidden state h. By default, the `hidden_size` will be equal to the `input_size` in order to propogate the dynamics.
- **num_layers** (*int, default=1*) – Number of recurrent layers. E.g., setting num_layers=2 would mean stacking two LSTMs together to form a stacked LSTM, with the second LSTM taking in outputs of the first LSTM and computing the final results.
- **bias** (*bool, default=True*) – If False, then the layer does not use bias weights b_ih and b_hh. Default: True
- **dropout** (*float, default=0.0*) – If non-zero, introduces a Dropout layer on the outputs of each LSTM layer except the last layer, with dropout probability equal to dropout.
- **bidirectional** (*bool, default=False*) – If True, becomes a bidirectional LSTM.

`forward`(*x: torch.Tensor*) → torch.Tensor

Parameters **x** (*torch.Tensor*) – Tensor of shape BxNxD for B batches of N examples by D feature dimensions.

Returns *torch.Tensor* – The predicted tensor of size (B, N, hidden_size).

`mse_loss`(*y_true: torch.Tensor*, *y_pred: torch.Tensor*, *reduction: str = 'mean'*) → torch.Tensor

Compute the MSE loss between `y_true` and `y_pred`.

Parameters

- **y_true** (*torch.Tensor*) – The true data.
- **y_pred** (*torch.Tensor*) – The prediction.
- **reduction** (*str, default="mean"*) – The reduction strategy for the F.mse_loss function.

Returns *torch.Tensor* – The MSE loss between `y_true` and `y_pred`.

`class mdlearn.nn.models.lstm.LSTMTrainer`(*input_size: int*, *hidden_size: Optional[int] = None*, *num_layers: int = 1*, *bias: bool = True*, *dropout: float = 0.0*, *bidirectional: bool = False*, *window_size: int = 10*, *horizon: int = 1*, *seed: int = 42*, *in_gpu_memory: bool = False*, *num_data_workers: int = 0*, *prefetch_factor: int = 2*, *split_pct: float = 0.8*, *split_method: str = 'partition'*, *batch_size: int = 128*, *shuffle: bool = True*, *device: str = 'cpu'*, *optimizer_name: str = 'RMSprop'*, *optimizer_hparams: Dict[str, Any] = {'lr': 0.001, 'weight_decay': 1e-05}*, *scheduler_name: Optional[str] = None*, *scheduler_hparams: Dict[str, Any] = {}*, *epochs: int = 100*, *verbose: bool = False*, *clip_grad_max_norm: float = 10.0*, *checkpoint_log_every: int = 10*, *plot_log_every: int = 10*, *plot_n_samples: int = 10000*, *plot_method: Optional[str] = 'TSNE'*, *train_subsample_pct: float = 1.0*, *valid_subsample_pct: float = 1.0*, *use_wandb: bool = False*)

Trainer class to fit an LSTM model to a time series of feature vectors.

`__init__`(*input_size: int*, *hidden_size: Optional[int] = None*, *num_layers: int = 1*, *bias: bool = True*, *dropout: float = 0.0*, *bidirectional: bool = False*, *window_size: int = 10*, *horizon: int = 1*, *seed: int = 42*, *in_gpu_memory: bool = False*, *num_data_workers: int = 0*, *prefetch_factor: int = 2*, *split_pct: float = 0.8*, *split_method: str = 'partition'*, *batch_size: int = 128*, *shuffle: bool = True*, *device: str = 'cpu'*, *optimizer_name: str = 'RMSprop'*, *optimizer_hparams: Dict[str, Any] = {'lr': 0.001, 'weight_decay': 1e-05}*, *scheduler_name: Optional[str] = None*, *scheduler_hparams: Dict[str, Any] = {}*, *epochs: int = 100*, *verbose: bool = False*, *clip_grad_max_norm: float = 10.0*, *checkpoint_log_every: int = 10*, *plot_log_every: int = 10*, *plot_n_samples: int = 10000*, *plot_method: Optional[str] = 'TSNE'*, *train_subsample_pct: float = 1.0*, *valid_subsample_pct: float = 1.0*, *use_wandb: bool = False*)

Parameters

- **input_size** (*int*) – The number of expected features in the input x.
- **hidden_size** (*Optional[int], default=None*) – The number of features in the hidden state h. By default, the `hidden_size` will be equal to the `input_size` in order to propogate the dynamics.
- **num_layers** (*int, default=1*) – Number of recurrent layers. E.g., setting num_layers=2 would mean stacking two LSTMs together to form a stacked LSTM, with the second LSTM taking in outputs of the first LSTM and computing the final results.
- **bias** (*bool, default=True*) – If False, then the layer does not use bias weights b_ih and b_hh. Default: True
- **dropout** (*float, default=0.0*) – If non-zero, introduces a Dropout layer on the outputs of each LSTM layer except the last layer, with dropout probability equal to dropout.
- **bidirectional** (*bool, default=False*) – If True, becomes a bidirectional LSTM.
- **window_size** (*int, default=10*) – Number of timesteps considered for prediction.
- **horizon** (*int, default=1*) – How many time steps to predict ahead.
- **seed** (*int, default=42*) – Random seed for torch, numpy, and random module.
- **in_gpu_memory** (*bool, default=False*) – If True, will pre-load the entire `data` array to GPU memory.
- **num_data_workers** (*int, default=0*) – How many subprocesses to use for data loading. 0 means that the data will be loaded in the main process.
- **prefetch_factor** (*int, by default=2*) – Number of samples loaded in advance by each worker. 2 means there will be a total of 2 * num_workers samples prefetched across all workers.
- **split_pct** (*float, default=0.8*) – Proportion of data set to use for training. The rest goes to validation.
- **split_method** (*str, default="random"*) – Method to split the data. For random split use "random", for a simple partition, use "partition".
- **batch_size** (*int, default=128*) – Mini-batch size for training.
- **shuffle** (*bool, default=True*) – Whether to shuffle training data or not.
- **device** (*str, default="cpu"*) – Specify training hardware either `cpu` or `cuda` for GPU devices.
- **optimizer_name** (*str, default="RMSprop"*) – Name of the PyTorch optimizer to use. Matches PyTorch optimizer class name.

- **optimizer_hparams** (*Dict[str, Any], default={"lr": 0.001, "weight_decay": 0.00001}*) – Dictionary of hyperparameters to pass to the chosen PyTorch optimizer.
- **scheduler_name** (*Optional[str], default=None*) – Name of the PyTorch learning rate scheduler to use. Matches PyTorch optimizer class name.
- **scheduler_hparams** (*Dict[str, Any], default={}*) – Dictionary of hyperparameters to pass to the chosen PyTorch learning rate scheduler.
- **epochs** (*int, default=100*) – Number of epochs to train for.
- **verbose** (*bool, default=False*) – If True, will print training and validation loss at each epoch.
- **clip_grad_max_norm** (*float, default=10.0*) – Max norm of the gradients for gradient clipping for more information see: `torch.nn.utils.clip_grad_norm_` documentation.
- **checkpoint_log_every** (*int, default=10*) – Epoch interval to log a checkpoint file containing the model weights, optimizer, and scheduler parameters.
- **plot_log_every** (*int, default=10*) – Epoch interval to log a visualization plot of the latent space.
- **plot_n_samples** (*int, default=10000*) – Number of validation samples to use for plotting.
- **plot_method** (*Optional[str], default="TSNE"*) – The method for visualizing the latent space or if visualization should not be run, set `plot_method=None`. If using `"TSNE"`, it will attempt to use the RAPIDS.ai GPU implementation and will fallback to the sklearn CPU implementation if RAPIDS.ai is unavailable.
- **train_subsample_pct** (*float, default=1.0*) – Percentage of training data to use during hyperparameter sweeps.
- **valid_subsample_pct** (*float, default=1.0*) – Percentage of validation data to use during hyperparameter sweeps.
- **use_wandb** (*bool, default=False*) – If True, will log results to wandb.

Raises

- **ValueError** – `split_pct` should be between 0 and 1.
- **ValueError** – `train_subsample_pct` should be between 0 and 1.
- **ValueError** – `valid_subsample_pct` should be between 0 and 1.
- **ValueError** – Specified `device` as `cuda`, but it is unavailable.

fit(*X: numpy.ndarray*, *scalars: Dict[str, numpy.ndarray] = {}*, *output_path: Union[str, pathlib.Path] = './'*, *checkpoint: Optional[Union[str, pathlib.Path]] = None*)

Trains the LSTM on the input data `X`.

Parameters

- **X** (*np.ndarray*) – Input features vectors of shape (N, D) where N is the number of data examples, and D is the dimension of the feature vector.
- **scalars** (*Dict[str, np.ndarray], default={}*) – Dictionary of scalar arrays. For instance, the root mean squared deviation (RMSD) for each feature vector can be passed via `{"rmsd": np.array(...)}`. The dimension of each scalar array should match the number of input feature vectors N.
- **output_path** (*PathLike, default="./"*) – Path to write training results to. Makes an `output_path/checkpoints` folder to save model checkpoint files, and `output_path/plots` folder to store latent space visualizations.

- **checkpoint** (*Optional[PathLike], default=None*) – Path to a specific model checkpoint file to restore training.

Raises

- **ValueError** – If `X` does not have two dimensions. For scalar time series, please reshape to (N, 1).
- **TypeError** – If `scalars` is not type dict. A common error is to pass `output_path` as the second argument.
- **NotImplementedError** – If using a learning rate scheduler other than `ReduceLROnPlateau`, a step function will need to be implemented.

predict(*X: numpy.ndarray*, *inference_batch_size: int = 512*, *checkpoint: Optional[Union[str, pathlib.Path]] = None*) → Tuple[numpy.ndarray, float]

Predict using the LSTM.

Parameters

- **X** (*np.ndarray*) – The input data to predict on.
- **inference_batch_size** (*int, default=512*) – The batch size for inference.
- **checkpoint** (*Optional[PathLike], default=None*) – Path to a specific model checkpoint file.

Returns *Tuple[np.ndarray, float]* – The predictions and the average MSE loss.

mdlearn.nn.models.vae

Modules

mdlearn.nn.models.vae.model
mdlearn.nn.models.vae.symmetric_conv2d_vae

mdlearn.nn.models.vae.model

Classes

VAE(*args, **kwargs)	Variational autoencoder base class module.

class mdlearn.nn.models.vae.model.**VAE**(**args: Any*, ***kwargs: Any*)

Variational autoencoder base class module. Inherits from `mdlearn.nn.models.ae.AE`.

__init__(*encoder*, *decoder*)

Parameters

- **encoder** (*torch.nn.Module*) – The encoder module.
- **decoder** (*torch.nn.Module*) – The decoder module.

encode(**args*, ***kwargs*) → torch.Tensor

Encoder forward pass and reparameterization of mu and logstd.

Parameters

- ***args** – Variable length encoder argument list.
- ****kwargs** – Arbitrary encoder keyword arguments.

Returns *torch.Tensor* – The encoded z-latent batch tensor.

Notes

Clamps logstd using a max logstd of 10.

kld_loss(*mu: Optional[torch.Tensor] = None*, *logstd: Optional[torch.Tensor] = None*) → torch.Tensor

Computes the KLD loss, either for the passed arguments `mu` and `logstd`, or based on latent variables from last encoding.

Parameters

- **mu** (*torch.Tensor, optional*) – The latent space for μ. If set to `None`, uses the last computation of μ.
- **logstd** (*torch.Tensor, optional*) – The latent space for $\log \sigma$. If set to `None`, uses the last computation of $\log \sigma^2$.

Returns *torch.Tensor* – KL divergence loss given `mu` and `logstd`.

Notes

Clamps logstd using a max logstd of 10.

reparametrize(*mu: torch.Tensor*, *logstd: torch.Tensor*) → torch.Tensor

Reparameterization trick for `mu` and `logstd`.

Parameters

- **mu** (*torch.Tensor*) – First encoder output.
- **logstd** (*torch.Tensor*) – Second encoder output.

Returns *torch.Tensor* – If training, return the reparametrized output. Otherwise, return `mu`.

mdlearn.nn.models.vae.symmetric_conv2d_vae

Classes

SymmetricConv2dVAE(*args, **kwargs)	Convolutional variational autoencoder from the "Deep clustering of protein folding simulations" paper.
SymmetricConv2dVAETrainer(input_shape[, ...])	Trainer class to fit a convolutional variational autoencoder to a set of contact maps.

class mdlearn.nn.models.vae.symmetric_conv2d_vae.**SymmetricConv2dVAE**(**args: Any*, ***kwargs: Any*)

Convolutional variational autoencoder from the "Deep clustering of protein folding simulations" paper. Inherits from `mdlearn.nn.models.vae.VAE`.

__init__(*input_shape: Tuple[int, int, int]*, *init_weights: Optional[str] = None*, *filters: List[int] = [64, 64, 64]*, *kernels: List[int] = [3, 3, 3]*, *strides: List[int] = [1, 2, 1]*, *affine_widths: List[int] = [128]*, *affine_dropouts: List[float] = [0.0]*, *latent_dim: int = 3*, *activation: str = 'ReLU'*, *output_activation: str = 'Sigmoid'*)

Parameters

- **input_shape** (*Tuple[int, int, int]*) – (1, height, width) input dimensions of input image.
- **init_weights** (*Optional[str]*) – .pt weights file to initial weights with.
- **filters** (*List[int]*) – Convolutional filter dimensions.
- **kernels** (*List[int]*) – Convolutional kernel dimensions (assumes square kernel).
- **strides** (*List[int]*) – Convolutional stride lengths (assumes square strides).
- **affine_widths** (*List[int]*) – Number of neurons in each linear layer.
- **affine_dropouts** (*List[float]*) – Dropout probability for each linear layer. Dropout value of 0.0 will skip adding the dropout layer.
- **latent_dim** (*int*) – Latent dimension for mu and $logstd$ layers.
- **activation** (*str*) – Activation function to use between convultional and linear layers.
- **output_activation** (*str*) – Output activation function for last decoder layer.

forward(*x: torch.Tensor*) → Tuple[torch.Tensor, torch.Tensor]

Forward pass of variational autoencoder.

Parameters **x** (*torch.Tensor*) – Input `x` data to encode and reconstruct.

Returns

- *torch.Tensor* – z-latent space batch tensor.
- *torch.Tensor* – `recon_x` reconstruction of `x`.

class mdlearn.nn.models.vae.symmetric_conv2d_vae.**SymmetricConv2dVAETrainer**(*input_shape: Tuple[int, int, int]*, *filters: List[int] = [64, 64, 64]*, *kernels: List[int] = [3, 3, 3]*, *strides: List[int] = [1, 2, 1]*, *affine_widths: List[int] = [128]*, *affine_dropouts: List[float] = [0.0]*, *latent_dim: int = 10*, *activation: str = 'ReLU'*, *output_activation: str = 'Sigmoid'*, *lambda_rec: float = 1.0*, *seed: int = 42*, *num_data_workers: int = 0*, *prefetch_factor: int = 2*, *split_pct: float = 0.8*, *split_method: str = 'random'*, *batch_size: int = 128*, *shuffle: bool = True*, *device: str = 'cpu'*, *optimizer_name: str = 'RMSprop'*, *optimizer_hparams: Dict[str, Any] = {'lr': 0.001, 'weight_decay': 1e-05}*, *scheduler_name: Optional[str] = None*, *scheduler_hparams: Dict[str, Any] = {}*, *epochs: int = 100*, *verbose: bool = False*, *clip_grad_max_norm: float = 10.0*, *checkpoint_log_every: int = 10*, *plot_log_every: int = 10*, *plot_n_samples: int = 10000*, *plot_method: Optional[str] = None*, *train_subsample_pct: float = 1.0*,

Trainer class to fit a convolutional variational autoencoder to a set of contact maps.

__init__(*input_shape: Tuple[int, int, int]*, *filters: List[int] = [64, 64, 64]*, *kernels: List[int] = [3, 3, 3]*, *strides: List[int] = [1, 2, 1]*, *affine_widths: List[int] = [128]*, *affine_dropouts: List[float] = [0.0]*, *latent_dim: int = 10*, *activation: str = 'ReLU'*, *output_activation: str = 'Sigmoid'*, *lambda_rec: float = 1.0*, *seed: int = 42*, *num_data_workers: int = 0*, *prefetch_factor: int = 2*, *split_pct: float = 0.8*, *split_method: str = 'random'*, *batch_size: int = 128*, *shuffle: bool = True*, *device: str = 'cpu'*, *optimizer_name: str = 'RMSprop'*, *optimizer_hparams: Dict[str, Any] = {'lr': 0.001, 'weight_decay': 1e-05}*, *scheduler_name: Optional[str] = None*, *scheduler_hparams: Dict[str, Any] = {}*, *epochs: int = 100*, *verbose: bool = False*, *clip_grad_max_norm: float = 10.0*, *checkpoint_log_every: int = 10*, *plot_log_every: int = 10*, *plot_n_samples: int = 10000*, *plot_method: Optional[str] = None*, *train_subsample_pct: float = 1.0*, *valid_subsample_pct: float = 1.0*, *use_wandb: bool = False*)

Parameters

- **input_shape** (*Tuple[int, int, int]*) – (1, height, width) input dimensions of input image.
- **filters** (*List[int], default=[64, 64, 64]*) – Convolutional filter dimensions.
- **kernels** (*List[int], default=[3, 3, 3]*) – Convolutional kernel dimensions (assumes square kernel).
- **strides** (*List[int], default=[1, 2, 1]*) – Convolutional stride lengths (assumes square strides).
- **affine_widths** (*List[int], default=[128]*) – Number of neurons in each linear layer. Defines the shape of the autoencoder (does not include latent dimension). The encoder and decoder are symmetric.
- **affine_dropouts** (*List[float], default=[0.0]*) – Dropout probability for each linear layer. Dropout value of 0.0 will skip adding the dropout layer.
- **latent_dim** (*int, default=10*) – Latent dimension for mu and $logstd$ layers.
- **activation** (*str, default="ReLU"*) – Activation function to use between convultional and linear layers.
- **output_activation** (*str, default="Sigmoid"*) – Output activation function for last decoder layer.
- **lambda_rec** (*float, default=1.0*) – Factor to scale reconstruction loss by during training such that `loss = lambda_rec * recon_loss + kld_loss`.
- **seed** (*int, default=42*) – Random seed for torch, numpy, and random module.
- **num_data_workers** (*int, default=0*) – How many subprocesses to use for data loading. 0 means that the data will be loaded in the main process.
- **prefetch_factor** (*int, by default=2*) – Number of samples loaded in advance by each worker. 2 means there will be a total of 2 * num_workers samples prefetched across all workers.
- **split_pct** (*float, default=0.8*) – Proportion of data set to use for training. The rest goes to validation.
- **split_method** (*str, default="random"*) – Method to split the data. For random split use "random", for a simple partition, use "partition".
- **batch_size** (*int, default=128*) – Mini-batch size for training.
- **shuffle** (*bool, default=True*) – Whether to shuffle training data or not.

- **device** (*str, default="cpu"*) – Specify training hardware either `cpu` or `cuda` for GPU devices.
- **optimizer_name** (*str, default="RMSprop"*) – Name of the PyTorch optimizer to use. Matches PyTorch optimizer class name.
- **optimizer_hparams** (*Dict[str, Any], default={"lr": 0.001, "weight_decay": 0.00001}*) – Dictionary of hyperparameters to pass to the chosen PyTorch optimizer.
- **scheduler_name** (*Optional[str], default=None*) – Name of the PyTorch learning rate scheduler to use. Matches PyTorch optimizer class name.
- **scheduler_hparams** (*Dict[str, Any], default={}*) – Dictionary of hyperparameters to pass to the chosen PyTorch learning rate scheduler.
- **epochs** (*int, default=100*) – Number of epochs to train for.
- **verbose** (*bool, default False*) – If True, will print training and validation loss at each epoch.
- **clip_grad_max_norm** (*float, default=10.0*) – Max norm of the gradients for gradient clipping for more information see: `torch.nn.utils.clip_grad_norm_` documentation.
- **checkpoint_log_every** (*int, default=10*) – Epoch interval to log a checkpoint file containing the model weights, optimizer, and scheduler parameters.
- **plot_log_every** (*int, default=10*) – Epoch interval to log a visualization plot of the latent space.
- **plot_n_samples** (*int, default=10000*) – Number of validation samples to use for plotting.
- **plot_method** (*Optional[str], default=None*) – The method for visualizing the latent space or if visualization should not be run, set `plot_method=None`. If using `"TSNE"`, it will attempt to use the RAPIDS.ai GPU implementation and will fallback to the sklearn CPU implementation if RAPIDS.ai is unavailable. A fast alternative is to plot the raw embeddings (or up to the first 3 dimensions if D > 3) using `"raw"`.
- **train_subsample_pct** (*float, default=1.0*) – Percentage of training data to use during hyperparameter sweeps.
- **valid_subsample_pct** (*float, default=1.0*) – Percentage of validation data to use during hyperparameter sweeps.
- **use_wandb** (*bool, default=False*) – If True, will log results to wandb. Metric keys include *"train_loss"*, *"train_recon_loss"*, *"train_kld_loss"*, *"valid_loss"*, *"valid_recon_loss"* and *"valid_kld_loss"*.

Raises

- **ValueError** – `split_pct` should be between 0 and 1.
- **ValueError** – `train_subsample_pct` should be between 0 and 1.
- **ValueError** – `valid_subsample_pct` should be between 0 and 1.
- **ValueError** – Specified `device` as `cuda`, but it is unavailable.

Examples

For an accompanying example, see: https://github.com/ramanathanlab/mdlearn/tree/main/examples/symmetric_conv2d_vae/training.

fit(*X: numpy.ndarray*, *scalars: Dict[str, numpy.ndarray] = {}*, *output_path: Union[str, pathlib.Path] = './'*, *checkpoint: Optional[Union[str, pathlib.Path]] = None*)

Trains the autoencoder on the input data `X`.

Parameters

- **X** (*np.ndarray*) – Input contact matrices in sparse COO format of shape (N,) where N is the number of data examples, and the empty dimension is ragged. The row and column index vectors should be contatenated and the values are assumed to be 1 and don't need to be explcitly passed.
- **scalars** (*Dict[str, np.ndarray], default={}*) – Dictionary of scalar arrays. For instance, the root mean squared deviation (RMSD) for each feature vector can be passed via `{"rmsd": np.array(...)}`. The dimension of each scalar array should match the number of input feature vectors N.
- **output_path** (*PathLike, default="./"*) – Path to write training results to. Makes an `output_path/checkpoints` folder to save model checkpoint files, and `output_path/plots` folder to store latent space visualizations.
- **checkpoint** (*Optional[PathLike], default=None*) – Path to a specific model checkpoint file to restore training.

Raises

- **TypeError** – If `scalars` is not type dict. A common error is to pass `output_path` as the second argument.
- **NotImplementedError** – If using a learning rate scheduler other than `ReduceLROnPlateau`, a step function will need to be implemented.

predict(*X: numpy.ndarray*, *inference_batch_size: int = 128*, *checkpoint: Optional[Union[str, pathlib.Path]] = None*) → Tuple[numpy.ndarray, float, float, float]

Predict using the LinearAE

Parameters

- **X** (*np.ndarray*) – Input contact matrices in sparse COO format of shape (N,) where N is the number of data examples, and the empty dimension is ragged. The row and column index vectors should be contatenated and the values are assumed to be 1 and don't need to be explcitly passed.
- **inference_batch_size** (*int, default=128*) – The batch size for inference.
- **checkpoint** (*Optional[PathLike], default=None*) – Path to a specific model checkpoint file.

Returns *Tuple[np.ndarray, float, float, float, float]* – The `z` latent vectors corresponding to the input data `X` and the average losses [total, reconstruction, KL-divergence]

mdlearn.nn.models.vde

Modules

---

mdlearn.nn.models.vde.model

> **Warning:** VDE models are still under development, use with caution!

---

mdlearn.nn.models.vde.symmetric_conv2d_vde

> **Warning:** VDE models are still under development, use with caution!

---

mdlearn.nn.models.vde.model

Warning: VDE models are still under development, use with caution!

Classes

VDE(*args, **kwargs)	Variational dynamics encoder base class module based off the "Variational Encoding of Complex Dynamics" paper Inherits from `mdlearn.nn.models.vae.VAE`.

class mdlearn.nn.models.vde.model.**VDE**(**args: Any*, ***kwargs: Any*)

Variational dynamics encoder base class module based off the "Variational Encoding of Complex Dynamics" paper Inherits from `mdlearn.nn.models.vae.VAE`.

ac_loss(*z_t: torch.Tensor*, *z_t_tau: torch.Tensor*) → torch.Tensor

Negative autocorrelation loss.

Parameters

- **z_t** (*torch.Tensor*) – z_t-latent vector.
- **z_t_tau** (*torch.Tensor*) – $z_{t+\tau}$-latent vector.

Returns *torch.Tensor* – Negative autocorrelation loss between z_t and $z_{t+\tau}$.

mdlearn.nn.models.vde.symmetric_conv2d_vde

Warning: VDE models are still under development, use with caution!

Classes

SymmetricConv2dVDE(*args, **kwargs)	Convolutional variational autoencoder from the "Deep clustering of protein folding simulations" paper implemented as a time lagged autoencoder.

class mdlearn.nn.models.vde.symmetric_conv2d_vde.**SymmetricConv2dVDE**(**args: Any*, ***kwargs: Any*)

Convolutional variational autoencoder from the "Deep clustering of protein folding simulations" paper implemented as a time lagged autoencoder. Inherits from `mdlearn.nn.models.vae.VDE`.

__init__(*input_shape: Tuple[int, ...]*, *init_weights: Optional[str] = None*, *filters: List[int] = [64, 64, 64]*, *kernels: List[int] = [3, 3, 3]*, *strides: List[int] = [1, 2, 1]*, *affine_widths: List[int] = [128]*, *affine_dropouts: List[float] = [0.0]*, *latent_dim: int = 3*, *activation: str = 'ReLU'*, *output_activation: str = 'Sigmoid'*)

Parameters

- **input_shape** (*Tuple[int, ...]*) – (height, width) input dimensions of input image.
- **init_weights** (*Optional[str]*) – .pt weights file to initial weights with.

- **filters** (*List[int]*) – Convolutional filter dimensions.
- **kernels** (*List[int]*) – Convolutional kernel dimensions (assumes square kernel).
- **strides** (*List[int]*) – Convolutional stride lengths (assumes square strides).
- **affine_widths** (*List[int]*) – Number of neurons in each linear layer.
- **affine_dropouts** (*List[float]*) – Dropout probability for each linear layer. Dropout value of 0.0 will skip adding the dropout layer.
- **latent_dim** (*int*) – Latent dimension for mu and $logstd$ layers.
- **activation** (*str*) – Activation function to use between convultional and linear layers.
- **output_activation** (*str*) – Output activation function for last decoder layer.

forward(*x: torch.Tensor*) → Tuple[torch.Tensor, torch.Tensor]

Forward pass of variational autoencoder.

Parameters **x** (*torch.Tensor*) – Input `x` data to encode and reconstruct.

Returns

- *torch.Tensor* – z-latent space batch tensor.
- *torch.Tensor* – `recon_x` reconstruction of `x`.

mdlearn.nn.models.wae

Modules

mdlearn.nn.models.wae.model
mdlearn.nn.models.wae.symmetric_conv2d_wae

mdlearn.nn.models.wae.model

Classes

WAE(*args, **kwargs)	Wasserstein autoencoder base class module.

class mdlearn.nn.models.wae.model.**WAE**(**args: Any*, ***kwargs: Any*)

Wasserstein autoencoder base class module. Inherits from `mdlearn.nn.models.vae.VAE`.

mmdrf_loss(*z: torch.Tensor*, *sigma: float*, *kernel: str*, *rf_dim: int*, *rf_resample: bool*) → torch.Tensor

Computes the loss $|\mu_{real} - \mu_{fake}|_H$

Parameters

- **z** (*torch.Tensor*) – The z-latent vector.
- **sigma** (*float*) – TODO
- **kernel** (*str*) – The type of kernel function to use.
- **rf_dim** (*int*) – Random features kernel dimension.

- **rf_resample** (*bool*) – Whether ot not to resample the random features.

Returns *torch.Tensor* – MMD RF loss.

mdlearn.nn.models.wae.symmetric_conv2d_wae

Classes

SymmetricConv2dWAE(*args, **kwargs)	Convolutional variational autoencoder from the "Deep clustering of protein folding simulations" paper implemented with wasserstein regularization.

class mdlearn.nn.models.wae.symmetric_conv2d_wae.**SymmetricConv2dWAE**(**args: Any*, ***kwargs: Any*)

Convolutional variational autoencoder from the "Deep clustering of protein folding simulations" paper implemented with wasserstein regularization. Inherits from `mdlearn.nn.models.wae.WAE`.

__init__(*input_shape: Tuple[int, ...]*, *init_weights: Optional[str] = None*, *filters: List[int] = [64, 64, 64]*, *kernels: List[int] = [3, 3, 3]*, *strides: List[int] = [1, 2, 1]*, *affine_widths: List[int] = [128]*, *affine_dropouts: List[float] = [0.0]*, *latent_dim: int = 3*, *activation: str = 'ReLU'*, *output_activation: str = 'Sigmoid'*)

Parameters

- **input_shape** (*Tuple[int, ...]*) – (height, width) input dimensions of input image.
- **init_weights** (*Optional[str]*) – .pt weights file to initial weights with.
- **filters** (*List[int]*) – Convolutional filter dimensions.
- **kernels** (*List[int]*) – Convolutional kernel dimensions (assumes square kernel).
- **strides** (*List[int]*) – Convolutional stride lengths (assumes square strides).
- **affine_widths** (*List[int]*) – Number of neurons in each linear layer.
- **affine_dropouts** (*List[float]*) – Dropout probability for each linear layer. Dropout value of 0.0 will skip adding the dropout layer.
- **latent_dim** (*int*) – Latent dimension for mu and $logstd$ layers.
- **activation** (*str*) – Activation function to use between convultional and linear layers.
- **output_activation** (*str*) – Output activation function for last decoder layer.

forward(*x: torch.Tensor*) → Tuple[torch.Tensor, torch.Tensor]

Forward pass of variational autoencoder.

Parameters **x** (*torch.Tensor*) – Input `x` data to encode and reconstruct.

Returns

- *torch.Tensor* – z-latent space batch tensor.
- *torch.Tensor* – `recon_x` reconstruction of `x`.

mdlearn.nn.modules

Modules

mdlearn.nn.modules.conv1d_encoder	Conv1dEncoder module for point cloud data.
mdlearn.nn.modules.conv2d_decoder	
mdlearn.nn.modules.conv2d_encoder	
mdlearn.nn.modules.dense_net	DenseNet module.
mdlearn.nn.modules.linear_decoder	LinearDecoder module for point cloud data.
mdlearn.nn.modules.linear_discriminator	
mdlearn.nn.modules.lstm_net	LSTMNet module.

mdlearn.nn.modules.conv1d_encoder

Conv1dEncoder module for point cloud data.

Classes

Conv1dEncoder(*args, **kwargs)	

class mdlearn.nn.modules.conv1d_encoder.**Conv1dEncoder**(**args: Any*, ***kwargs: Any*)

__init__(*num_points: int*, *num_features: int = 0*, *latent_dim: int = 20*, *bias: bool = True*, *relu_slope: float = 0.0*, *filters: List[int] = [64, 128, 256, 256, 512]*, *kernels: List[int] = [5, 5, 3, 1, 1]*)

Conv1dEncoder module for point cloud data.

Parameters

- **num_points** (*int*) – Number of input points in point cloud.
- **num_features** (*int, optional*) – Number of scalar features per point in addition to 3D coordinates, by default 0.
- **latent_dim** (*int, optional*) – Latent dimension of the encoder, by default 20.
- **bias** (*bool, optional*) – Use a bias term in the Conv1d layers, by default True.
- **relu_slope** (*float, optional*) – If greater than 0.0, will use LeakyReLU activiation with `negative_slope` set to `relu_slope`, by default 0.0.
- **filters** (*List[int], optional*) – Encoder Conv1d filter sizes, by default [64, 128, 256, 256, 512].
- **kernels** (*List[int], optional*) – Encoder Conv1d kernel sizes, by default [5, 5, 3, 1, 1].

forward(*x: torch.Tensor*) → Tuple[torch.Tensor, torch.Tensor]

mdlearn.nn.modules.conv2d_decoder

Classes

Conv2dDecoder(*args, **kwargs)	

class mdlearn.nn.modules.conv2d_decoder.**Conv2dDecoder**(**args: Any*, ***kwargs: Any*)

forward(*x*)

init_weights(*init_weights: Optional[str]*)

mdlearn.nn.modules.conv2d_encoder

Classes

Conv2dEncoder(*args, **kwargs)	

class mdlearn.nn.modules.conv2d_encoder.**Conv2dEncoder**(**args: Any*, ***kwargs: Any*)

forward(*x: torch.Tensor*)

init_weights(*init_weights: Optional[str]*)

mdlearn.nn.modules.dense_net

DenseNet module.

Classes

DenseNet(*args, **kwargs)	

class mdlearn.nn.modules.dense_net.**DenseNet**(**args: Any*, ***kwargs: Any*)

__init__(*input_dim: int*, *neurons: List[int] = [128]*, *bias: bool = True*, *relu_slope: float = 0.0*, *inplace_activation: bool = False*)

DenseNet module for easy feedforward network creation. Creates a neural network with Linear layers and ReLU (or LeakyReLU activation). The returned tensor from the forward function, does not pass through an activation function.

Parameters

- **input_dim** (*int*) – Dimension of input tensor (should be flattened).
- **neurons** (*List[int], default=[128]*) – Linear layers `in_features`.
- **bias** (*bool, default=True*) – Use a bias term in the Linear layers.

- **relu_slope** (*float, default=0.0*) – If greater than 0.0, will use LeakyReLU activiation with `negative_slope` set to `relu_slope`.
- **inplace_activation** (*bool, default=False*) – Sets the inplace option for the activation function.

Raises `ValueError` – `neurons` should specify atleast one layer.

forward(*x: torch.Tensor*) → torch.Tensor

Forward pass through dense network.

Parameters **x** (*torch.Tensor*) – Input data.

Returns *torch.Tensor* – The output of the neural network with dimension (batch size, last neuron size).

mdlearn.nn.modules.linear_decoder

LinearDecoder module for point cloud data.

Classes

LinearDecoder(*args, **kwargs)	

class mdlearn.nn.modules.linear_decoder.**LinearDecoder**(**args: Any*, ***kwargs: Any*)

__init__(*num_points: int*, *num_features: int = 0*, *latent_dim: int = 20*, *bias: bool = True*, *relu_slope: float = 0.0*, *affine_widths: List[int] = [64, 128, 512, 1024]*)

LinearDecoder module for point cloud data.

Parameters

- **num_points** (*int*) – Number of input points in point cloud.
- **num_features** (*int, optional*) – Number of scalar features per point in addition to 3D coordinates, by default 0.
- **latent_dim** (*int, optional*) – Latent dimension of the decoder, by default 20.
- **bias** (*bool, optional*) – Use a bias term in the Linear layers, by default True.
- **relu_slope** (*float, optional*) – If greater than 0.0, will use LeakyReLU activiation with `negative_slope` set to `relu_slope`, by default 0.0.
- **affine_widths** (*List[int], optional*) – Linear layers `in_features`, by default [64, 128, 512, 1024].

forward(*z: torch.Tensor*) → torch.Tensor

mdlearn.nn.modules.linear_discriminator

Classes

LinearDiscriminator(*args, **kwargs)	

class mdlearn.nn.modules.linear_discriminator.**LinearDiscriminator**(**args: Any*, ***kwargs: Any*)

__init__(*latent_dim: int = 20*, *bias: bool = True*, *relu_slope: float = 0.0*, *affine_widths: List[int] = [512, 128, 64]*)

LinearDiscriminator module.

Parameters

- **latent_dim** (*int, optional*) – Latent dimension of the decoder, by default 20.
- **bias** (*bool, optional*) – Use a bias term in the Linear layers, by default True.
- **relu_slope** (*float, optional*) – If greater than 0.0, will use LeakyReLU activiation with `negative_slope` set to `relu_slope`, by default 0.0.
- **affine_widths** (*List[int], optional*) – Linear layers `in_features`, by default [64, 128, 512, 1024].

forward(*x: torch.Tensor*)

mdlearn.nn.modules.lstm_net

LSTMNet module.

Classes

LSTMNet(*args, **kwargs)	

class mdlearn.nn.modules.lstm_net.**LSTMNet**(**args: Any*, ***kwargs: Any*)

__init__(*input_dim: int*, *neurons: List[int] = [32]*, *bias: bool = True*, *dropout: float = 0.0*)

LSTMNet module for easy StackedLSTM network creation. The returned tensor from the forward function, is the hidden state of the final LSTM layer.

Parameters

- **input_dim** (*int*) – Dimension D of input tensor (N, D) where N is the length of the sequence and D is the dimension of each example.
- **neurons** (*List[int], default=[32]*) – LSTM layers `hidden_size`.
- **bias** (*bool, default=True*) – If False, then each layer does not use bias weights b_ih and b_hh.
- **dropout** (*float, default=0.0*) – If non-zero, introduces a Dropout layer on the outputs of each LSTM layer except the last layer, with dropout probability equal to dropout.

Note: Bidirectional LSTMs are not currently supported in this module.

Raises `ValueError` – `neurons` should specify atleast one layer.

`forward`(*x: torch.Tensor*) → torch.Tensor

Forward pass through LSTM network.

Parameters **x** (*torch.Tensor*) – Input data of shape (B, N, D) where B is the batch size, N is the length of the sequence, and D is the dimension.

Returns *torch.Tensor* – The output of the neural network with dimension (batch size, last neuron size).

mdlearn.nn.utils

Functions

conv_output_dim(input_dim, kernel_size, ...)	**Parameters** • **input_dim** (*int*) -- input size. may include padding
conv_output_shape(input_dim, kernel_size, ...)	**Parameters** • **input_dim** (*tuple*) -- (height, width) dimensions for convolution input
get_activation(activation, *args, **kwargs)	**Parameters** **activation** (*str*) -- type of activation e.g. 'ReLU', etc
reset(nn)	
same_padding(input_dim, kernel_size, stride)	Returns Keras-like same padding.

Classes

Trainer([seed, in_gpu_memory, ...])	Trainer base class which implements training utility functions.

class mdlearn.nn.utils.**Trainer**(*seed: int = 42*, *in_gpu_memory: bool = False*, *num_data_workers: int = 0*, *prefetch_factor: int = 2*, *split_pct: float = 0.8*, *split_method: str = 'random'*, *batch_size: int = 128*, *shuffle: bool = True*, *device: str = 'cpu'*, *epochs: int = 100*, *verbose: bool = False*, *clip_grad_max_norm: float = 10.0*, *checkpoint_log_every: int = 10*, *plot_log_every: int = 10*, *plot_n_samples: int = 10000*, *plot_method: Optional[str] = 'TSNE'*, *train_subsample_pct: float = 1.0*, *valid_subsample_pct: float = 1.0*, *use_wandb: bool = False*)

Trainer base class which implements training utility functions.

`__init__`(*seed: int = 42*, *in_gpu_memory: bool = False*, *num_data_workers: int = 0*, *prefetch_factor: int = 2*, *split_pct: float = 0.8*, *split_method: str = 'random'*, *batch_size: int = 128*, *shuffle: bool = True*, *device: str = 'cpu'*, *epochs: int = 100*, *verbose: bool = False*, *clip_grad_max_norm: float = 10.0*, *checkpoint_log_every: int = 10*, *plot_log_every: int = 10*, *plot_n_samples: int = 10000*, *plot_method: Optional[str] = 'TSNE'*, *train_subsample_pct: float = 1.0*, *valid_subsample_pct: float = 1.0*, *use_wandb: bool = False*)

Parameters

- **seed** (*int, default=42*) – Random seed for torch, numpy, and random module.
- **in_gpu_memory** (*bool, default=False*) – If True, will pre-load the entire `data` array to GPU memory.
- **num_data_workers** (*int, default=0*) – How many subprocesses to use for data loading. 0 means that the data will be loaded in the main process.
- **prefetch_factor** (*int, by default=2*) – Number of samples loaded in advance by each worker. 2 means there will be a total of 2 * num_workers samples prefetched across all workers.
- **split_pct** (*float, default=0.8*) – Proportion of data set to use for training. The rest goes to validation.
- **split_method** (*str, default="random"*) – Method to split the data. For random split use "random", for a simple partition, use "partition".
- **batch_size** (*int, default=128*) – Mini-batch size for training.
- **shuffle** (*bool, default=True*) – Whether to shuffle training data or not.
- **device** (*str, default="cpu"*) – Specify training hardware either `cpu` or `cuda` for GPU devices.
- **epochs** (*int, default=100*) – Number of epochs to train for.
- **verbose** (*bool, default False*) – If True, will print training and validation loss at each epoch.
- **clip_grad_max_norm** (*float, default=10.0*) – Max norm of the gradients for gradient clipping for more information see: `torch.nn.utils.clip_grad_norm_` documentation.
- **checkpoint_log_every** (*int, default=10*) – Epoch interval to log a checkpoint file containing the model weights, optimizer, and scheduler parameters.
- **plot_log_every** (*int, default=10*) – Epoch interval to log a visualization plot of the latent space.
- **plot_n_samples** (*int, default=10000*) – Number of validation samples to use for plotting.
- **plot_method** (*Optional[str], default="TSNE"*) – The method for visualizing the latent space or if visualization should not be run, set `plot_method=None`. If using `"TSNE"`, it will attempt to use the RAPIDS.ai GPU implementation and will fallback to the sklearn CPU implementation if RAPIDS.ai is unavailable.
- **train_subsample_pct** (*float, default=1.0*) – Percentage of training data to use during hyperparameter sweeps.
- **valid_subsample_pct** (*float, default=1.0*) – Percentage of validation data to use during hyperparameter sweeps.
- **use_wandb** (*bool, default=False*) – If True, will log results to wandb.

Raises

- **`ValueError`** – `split_pct` should be between 0 and 1.

- **ValueError** – `train_subsample_pct` should be between 0 and 1.
- **ValueError** – `valid_subsample_pct` should be between 0 and 1.
- **ValueError** – Specified `device` as `cuda`, but it is unavailable.

Note: This base class does not receive optimizer or scheduler settings because in general there could be multiple optimizers.

---

fit()

Trains the model on the input dataset.

Raises **NotImplementedError** – Child class must implement this method.

predict()

Predicts using the trained model.

Raises **NotImplementedError** – Child class must implement this method.

step_scheduler(*epoch: int*, *avg_train_loss: float*, *avg_valid_loss: float*)

Implements the logic to step the learning rate scheduler. Different schedulers may have different update logic. Please subclass `LinearAETrainer` and re-implement this function for support of additional logic.

Parameters

- **epoch** (*int*) – The current training epoch.
- **avg_train_loss** (*float*) – The current epochs average training loss.
- **avg_valid_loss** (*float*) – The current epochs average valiation loss.

Raises **NotImplementedError** – If using a learning rate scheduler other than `ReduceLROnPlateau`, a step function will need to be added.

mdlearn.nn.utils.**conv_output_dim**(*input_dim*, *kernel_size*, *stride*, *padding*, *transpose=False*)

Parameters

- **input_dim** (*int*) – input size. may include padding
- **kernel_size** (*int*) – filter size
- **stride** (*int*) – stride length
- **padding** (*int*) – length of 0 pad

mdlearn.nn.utils.**conv_output_shape**(*input_dim*, *kernel_size*, *stride*, *padding*, *num_filters*, *transpose=False*, *dim=2*)

Parameters

- **input_dim** (*tuple*) – (height, width) dimensions for convolution input
- **kernel_size** (*int*) – filter size
- **stride** (*int*) – stride length
- **padding** (*tuple*) – (height, width) length of 0 pad
- **num_filters** (*int*) – number of filters
- **transpose** (*bool*) – signifies whether Conv or ConvTranspose
- **dim** (*int*) – 1 or 2, signifies Conv1d or Conv2d

Returns *(channels, height, width) tuple*

mdlearn.nn.utils.**get_activation**(*activation*, **args*, ***kwargs*)

Parameters **activation** (*str*) – type of activation e.g. 'ReLU', etc

mdlearn.nn.utils.**reset**(*nn*)

mdlearn.nn.utils.**same_padding**(*input_dim: Union[int, Tuple[int, int]]*, *kernel_size: int*, *stride: int*) → Union[int, Tuple[int, int]]

Returns Keras-like same padding. Works for rectangular input_dim.

Parameters

- **input_dim** (*tuple, int*) – (height, width) dimensions for Conv2d input int for Conv1d input
- **kernel_size** (*int*) – filter size
- **stride** (*int*) – stride length

Returns

- *int* – height of padding
- *int* – width of padding

mdlearn.utils

Configurations and utilities for model building and training.

Functions

get_torch_optimizer(name, hparams, parameters)	Construct a PyTorch optimizer specified by `name` and `hparams`.
get_torch_scheduler(name, hparams, optimizer)	Construct a PyTorch lr_scheduler specified by `name` and `hparams`.
log_checkpoint(checkpoint_file, epoch, ...)	Write a torch .pt file containing the epoch, model, optimizer, and scheduler.
parse_args()	Parse command line arguments using argparse library
resume_checkpoint(checkpoint_file, model, ...)	Modifies `model`, `optimizer`, and `scheduler` with values stored in torch .pt file `checkpoint_file` to resume from a previous training checkpoint.

pydantic settings mdlearn.utils.**BaseSettings**


```
{
   "title": "BaseSettings",
   "description": "Base class for settings, allowing values to be overridden by␣
↪environment variables.\n\nThis is useful in production for secrets you do not␣
↪wish to save in code, it plays nicely with docker(-compose),\nHeroku and any 12␣
↪factor app design.",
   "type": "object",
   "properties": {},
   "additionalProperties": false
}
```

dump_yaml(*cfg_path: Union[str, pathlib.Path]*)

classmethod from_yaml(*filename: Union[str, pathlib.Path]*) → mdlearn.utils._T

pydantic settings mdlearn.utils.**OptimizerConfig**

pydantic schema for PyTorch optimizer which allows for arbitrary optimizer hyperparameters.


```
{
   "title": "OptimizerConfig",
   "description": "pydantic schema for PyTorch optimizer which allows\nfor␣
↪arbitrary optimizer hyperparameters.",
   "type": "object",
   "properties": {
      "name": {
         "title": "Name",
         "default": "Adam",
         "env_names": "{'name'}",
         "type": "string"
      },
      "hparams": {
         "title": "Hparams",
         "default": {},
         "env_names": "{'hparams'}",
         "type": "object"
      }
   }
}
```


Config

- **extra**: *str = allow*

Fields

- *hparams (Dict[str, Any])*
- *name (str)*

field hparams: Dict[str, Any] = {}

field name: str = 'Adam'

pydantic settings mdlearn.utils.**SchedulerConfig**

pydantic schema for PyTorch scheduler which allows for arbitrary scheduler hyperparameters.


```
{
   "title": "SchedulerConfig",
   "description": "pydantic schema for PyTorch scheduler which allows for arbitrary\
↪nscheduler hyperparameters.",
   "type": "object",
   "properties": {
      "name": {
         "title": "Name",
         "default": "ReduceLROnPlateau",
         "env_names": "{'name'}",
         "type": "string"
```


(continues on next page)

(continued from previous page)


```
        },
        "hparams": {
            "title": "Hparams",
            "default": {},
            "env_names": "{'hparams'}",
            "type": "object"
        }
    }
}
```


Config

- **extra**: *str = allow*

Fields

- *hparams (Dict[str, Any])*
- *name (str)*

field hparams: Dict[str, Any] = {}

field name: str = 'ReduceLROnPlateau'

pydantic settings mdlearn.utils.**WandbConfig**


```
{
    "title": "WandbConfig",
    "description": "Base class for settings, allowing values to be overridden by␣
↪environment variables.\n\nThis is useful in production for secrets you do not␣
↪wish to save in code, it plays nicely with docker(-compose),\nHeroku and any 12␣
↪factor app design.",
    "type": "object",
    "properties": {
        "wandb_project_name": {
            "title": "Wandb Project Name",
            "env_names": "{'wandb_project_name'}",
            "type": "string"
        },
        "wandb_entity_name": {
            "title": "Wandb Entity Name",
            "env_names": "{'wandb_entity_name'}",
            "type": "string"
        },
        "model_tag": {
            "title": "Model Tag",
            "env_names": "{'model_tag'}",
            "type": "string"
        }
    },
    "additionalProperties": false
}
```


Fields

- *model_tag (Optional[str])*
- *wandb_entity_name (Optional[str])*
- *wandb_project_name (Optional[str])*

field model_tag: Optional[str] = None

field wandb_entity_name: Optional[str] = None

field wandb_project_name: Optional[str] = None

init(*cfg:* mdlearn.utils.BaseSettings, *model: torch.nn.Module*, *wandb_path: Union[str, pathlib.Path]*) → Optional[wandb.config]

Initialize wandb with model and config.

Parameters

- **cfg** (*BaseSettings*) – Model configuration with hyperparameters and training settings.
- **model** (*torch.nn.Module*) – Model to train, passed to `wandb.watch(model)` for logging.
- **wandb_path** (*PathLike*) – Path to write `wandb/` directory containing training logs.

Returns *Optional[wandb.config]* – wandb config object or None if *wandb_project_name* is None.

mdlearn.utils.**get_torch_optimizer**(*name: str*, *hparams: Dict[str, Any]*, *parameters*) → torch.optim.Optimizer

Construct a PyTorch optimizer specified by `name` and `hparams`.

mdlearn.utils.**get_torch_scheduler**(*name: Optional[str]*, *hparams: Dict[str, Any]*, *optimizer: torch.optim.Optimizer*) → Optional[torch.optim.lr_scheduler._LRScheduler]

Construct a PyTorch lr_scheduler specified by `name` and `hparams`.

Parameters

- **name** (*Optional[str]*) – Name of PyTorch lr_scheduler class to use. If `name` is `None`, simply return None.
- **hparams** (*Dict[str, Any]*) – Hyperparameters to pass to the lr_scheduler.
- **optimizer** (*torch.optim.Optimizer*) – The initialized optimizer.

Returns *Optional[torch.optim.lr_scheduler._LRScheduler]* – The initialized PyTorch scheduler, or None if `name` is `None`.

mdlearn.utils.**log_checkpoint**(*checkpoint_file: Union[str, pathlib.Path]*, *epoch: int*, *model: torch.nn.Module*, *optimizers: Dict[str, torch.optim.Optimizer]*, *scheduler: Optional[torch.optim.lr_scheduler._LRScheduler] = None*)

Write a torch .pt file containing the epoch, model, optimizer, and scheduler.

Parameters

- **checkpoint_file** (*PathLike*) – Path to save checkpoint file.
- **epoch** (*int*) – The current training epoch.
- **model** (*torch.nn.Module*) – The model whose parameters are saved.
- **optimizers** (*Dict[str, torch.optim.Optimizer]*) – The optimizers whose parameters are saved.

- **scheduler** (*Optional[torch.optim.lr_scheduler._LRScheduler]*) – Optional scheduler whose parameters are saved.

mdlearn.utils.**parse_args**() → argparse.Namespace

Parse command line arguments using argparse library

Returns *argparse.Namespace* – Dict like object containing a path to a YAML file accessed via the config property.

Example

```
>>> from mdlearn.utils import parse_args
>>> args = parse_args()
>>> # MyConfig should inherit from BaseSettings
>>> cfg = MyConfig.from_yaml(args.config)
```

mdlearn.utils.**resume_checkpoint**(*checkpoint_file: Union[str, pathlib.Path]*, *model: torch.nn.Module*, *optimizers: Dict[str, torch.optim.Optimizer]*, *scheduler: Optional[torch.optim.lr_scheduler._LRScheduler] = None*) → int

Modifies `model`, `optimizer`, and `scheduler` with values stored in torch .pt file `checkpoint_file` to resume from a previous training checkpoint.

Parameters

- **checkpoint_file** (*PathLike*) – Path to checkpoint file to resume from.
- **model** (*torch.nn.Module*) – Module to update the parameters of.
- **optimizers** (*Dict[str, torch.optim.Optimizer]*) – Optimizers to update.
- **scheduler** (*Optional[torch.optim.lr_scheduler._LRScheduler]*) – Optional scheduler to update.

Returns *int* – The epoch the checkpoint is saved plus one i.e. the current training epoch to start from.

mdlearn.visualize

Functions to visualize modeling results.

Functions

log_latent_visualization(data, colors, ...)	Make scatter plots of the latent space using the specified method of dimensionality reduction.
plot_scatter(data[, color_dict, color])	

mdlearn.visualize.**log_latent_visualization**(*data: numpy.ndarray*, *colors: Dict[str, numpy.ndarray]*, *output_path: Union[str, pathlib.Path]*, *epoch: int = 0*, *n_samples: Optional[int] = None*, *method: str = 'raw'*) → Dict[str, str]

Make scatter plots of the latent space using the specified method of dimensionality reduction.

Parameters

- **data** (*np.ndarray*) – The latent embeddings to visualize of shape (N, D) where N is the number of examples and D is the number of dimensions.
- **colors** (*Dict[str, np.ndarray]*) – Each item in the dictionary will generate a different plot labeled with the key name. Each inner array should be of size N.
- **output_path** (*PathLike*) – The output directory path to save plots to.
- **epoch** (*int, default=0*) – The current epoch of training to label plots with.
- **n_samples** (*Optional[int], default=None*) – Number of samples to plot, will take a random sample of the `data` if `n_samples < N`. Otherwise, if `n_samples` is None, use all the data.
- **method** (*str, default="raw"*) – Method of dimensionality reduction used to plot. Currently supports: "PCA", "TSNE", "LLE", or "raw" for plotting the raw embeddings (or up to the first 3 dimensions if D > 3). If "TSNE" is specified, then the GPU accelerated RAPIDS.ai implementation will be tried first and if it is unavailable then the sklearn version will be used instead.

Returns *Dict[str, str]* – A dictionary mapping each key in color to a raw HTML string containing the scatter plot data. These can be saved directly for visualization and logged to wandb during training.

Raises `ValueError` – If dimensionality reduction `method` is not supported.

`mdlearn.visualize.`**`plot_scatter`**(*data: numpy.ndarray*, *color_dict: Dict[str, numpy.ndarray] = {}*, *color: Optional[str] = None*) → plotly.graph_objects._figure.Figure

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

m

INDEX

Symbols

A

B

C

M

N

P

R

S

T

V

W