# PyKale

*Release 0.1.2*

**PyKale Contributors**

**Jul 13, 2023**

# GETTING STARTED

CHAPTER

ONE

# INTRODUCTION

PyKale is a **Py**thon library for **k**nowledge-**a**ware machine **le**arning from multiple sources, particularly from multiple modalities for multimodal learning and from multiple domains for transfer learning. This library was motivated by needs in healthcare applications (hence we choose the acronym *kale*, a healthy vegetable) and aims to enable and accelerate interdisciplinary research.

## 1.1 Objectives

Our objectives are to build *green* machine learning systems.

- *Reduce repetition and redundancy*: refactor code to standardize workflow and enforce styles, and identify and remove duplicated functionalities
- *Reuse existing resources*: reuse the same machine learning pipeline for different data, and reuse existing libraries for available functionalities
- *Recycle learning models across areas*: identify commonalities between applications, and recycle models for one application to another

## 1.2 API design

To achieve the above objectives, we

- design our API to be pipeline-based to unify the workflow and increase the flexibility, and
- follow core principles of standardization and minimalism in the development.

This design helps us break barriers between different areas or applications and facilitate the fusion and nurture of ideas across discipline boundaries.

## 1.3 Development

We have Research Software Engineers (RSEs) on our team to help us adopt the best software engineering practices in a research context. We have modern GitHub setup with project boards, discussion, and GitHub actions/workflows. Our repository has automation and continuous integration to build documentation, do linting of our code, perform pre-commit checks (e.g. maximum file size), use pytest for testing and codecov for analysis of testing.

CHAPTER

TWO

# INSTALLATION

## 2.1 Requirements

PyKale requires Python 3.8, 3.9, or 3.10. Before installing pykale, you should

- manually install PyTorch matching your hardware first,
- if you will use APIs related to graphs, you need to manually install PyTorch Geometric first following its official instructions and matching your PyTorch installation, and
- If RDKit will be used, you need to install it via `pip install rdkit`.

## 2.2 Pip install

Install PyKale using `pip` for the stable version:

```
pip install pykale  # for the core API only
```

## 2.3 Install from source

Install from source for the latest version and/or development:

```
git clone https://github.com/pykale/pykale
cd pykale
pip install .  # for the core API only
pip install -e .[dev]  # editable install for developers including all dependencies and␣
↪examples
```

## 2.4 Installation options

PyKale provides six installation options for different user needs:

- `default`: `pip install pykale` for essential functionality
- `graph`: `pip install pykale[graph]` for graph-related functionality (e.g., TDC)
- `image`: `pip install pykale[image]` for image-related functionality (e.g., DICOM)
- `example`: `pip install pykale[example]` for examples and tutorials

- `full`: `pip install pykale[full]` for all functionality, including examples and tutorials
- `dev`: `pip install pykale[dev]` for development, including all functionality, examples, and tutorials

Multiple options can be chosen by separating them with commas (without whitespace). See examples below.

```
pip install pykale[graph,example]
pip install pykale[graph,image]
pip install pykale[graph,image,example]
```

## 2.5 Tests

For local unit tests on all `kale` API, you need to have PyTorch, PyTorch Geometric, and RDKit installed (see the top) and then run pytest at the root directory:

```
pytest
```

You can also run pytest on individual module (see pytest documentation).

CHAPTER

THREE

# TUTORIAL

For *interactive* tutorials, see *Jupyter Notebook tutorials*.

## 3.1 Usage of Pipeline-based API in Examples

The `kale` API has a unique pipeline-based API design. Each example typically has three essential modules (`main.py`, `config.py`, `model.py`), one optional directory (`configs`), and possibly other modules (`trainer.py`):

- `main.py` is the main module to be run, showing the main workflow.
- `config.py` is the configuration module that sets up the data, prediction problem, and hyper-parameters, etc. The settings in this module is the default configuration.
  - `configs` is the directory to place *customized* configurations for individual runs. We use `.yaml` files for this purpose.
- `model.py` is the model module to define the machine learning model and configure its training parameters.
  - `trainer.py` is the trainer module to define the training and testing workflow. This module is *only needed when NOT using `PyTorch Lightning`*.

Next, we explain the usage of the pipeline-based API in the modules above, mainly using the domain adaptation for digits classification example.

- The `kale.pipeline` module provides mature, off-the-shelf machine learning pipelines for plug-in usage, e.g. `import kale.pipeline.domain_adapter as domain_adapter` in `digits_dann`'s `model` module.
- The `kale.utils` module provides common utility functions, such as `from kale.utils.seed import set_seed` in `digits_dann`'s `main` module.
- The `kale.loaddata` module provides the input to the machine learning system, such as `from kale.loaddata.image_access import DigitDatase` in `digits_dann`'s `main` module.
- The `kale.prepdata` module provides pre-processing functions to transform the raw input data into a suitable form for machine learning, such as `import kale.prepdata.image_transform as image_transform` in `kale.loaddata.image_access` used in `digits_dann`'s `main` module for image data augmentation.
- The `kale.embed` module provides *embedding* functions (the *encoder*) to *learn* suitable representations from the (pre-processed) input data, such as `from kale.embed.image_cnn import SmallCNNFeature` in `digits_dann`'s `model` module. This is a machine learning module.
- The `kale.predict` module provides prediction functions (the *decoder*) to *learn* a mapping from the input representation to a target prediction, such as `from kale.predict.class_domain_nets import ClassNetSmallImage` in `digits_dann`'s `model` module. This is also a machine learning module.
- The `kale.evaluate` module implements evaluation metrics not yet available, such as the Concordance Index (CI) for measuring the proportion of concordant pairs.

- The `kale.interpret` module aims to provide functions for interpretation of the learned model or the prediction results, such as visualization. This module has no implementation yet.

## 3.2 Building New Modules or Projects

New modules/projects can be built following the steps below.

- Step 1 - Examples: Choose one of the examples of your interest (e.g., most relevant to your project) to
  - browse through the configuration, main, and model modules
  - download the data if needed
  - run the example following instructions in the example's README
- Step 2a - New model: To develop new machine learning models under PyKale,
  - define the blocks in your pipeline to figure out what the methods are for data loading, pre-processing data, embedding (encoder/representation), prediction (decoder), evaluation, and interpretation (if needed)
  - modify existing pipelines with your customized blocks or build a new pipeline with PyKale blocks and blocks from other libraries
- Step 2b - New applications: To develop new applications using PyKale,
  - clarify the input data and the prediction target to find matching functionalities in PyKale (request if not found)
  - tailor data loading, pre-processing, and evaluation (and interpretation if needed) to your application

## 3.3 The Scope of Support

### 3.3.1 Data

PyKale currently supports graphs, images, and videos, using PyTorch Dataloaders wherever possible. Audios are not supported yet (welcome your contribution).

### 3.3.2 Machine learning models

PyKale supports modules from the following areas of machine learning

- Deep learning: convolutional neural networks (CNNs), graph neural networks (GNNs) GNN including graph convolutional networks (GCNs), transformers
- Transfer learning: domain adaptation
- Multimodal learning: integration of heterogeneous data
- Dimensionality reduction: multilinear subspace learning, such as multilinear principal component analysis (MPCA)

### 3.3.3 Example applications

PyKale includes example application from three areas below

- Image/video recognition: imaging recognition with CIFAR10/100, digits (MNIST, USPS, SVHN), action videos (EPIC Kitchen)
- Bioinformatics/graph analysis: link prediction problems in BindingDB and knowledge graphs
- Medical imaging: cardiac MRI classification

CHAPTER

FOUR

# JUPYTER NOTEBOOK TUTORIALS

- **Autism detection**: Domain adaptation on multi-site imaging data for autism detection
- **Cardiovascular disease diagnosis**: Cardiac MRI for MPCA-based diagnosis
- **Drug discovery**: BindingDB drug-target interaction prediction
- **Image classification**: Digits domain adaptation
- **Toy data classification**: Toy data domain adaptation

CHAPTER

FIVE

# CONFIGURATION USING YAML

## 5.1 Why YAML?

PyKale has been designed such that users can configure machine learning models and experiments without writing any new Python code. This is achieved via a human and machine readable language called YAML. Well thought out default configuration values are first stored using the YACS Python module in a `config.py` file. Several customized configurations can then be created in respective `.yaml` files.

This also enables more advanced users to establish their own default and add new configuration parameters with minimal coding. By separating code and configuration, this approach can lead to better reproducibility.

## 5.2 A simple example

The following example is a simple YAML file `tutorial.yaml` used by the digits tutorial notebook:

```
DAN:
  METHOD: "CDAN"

DATASET:
  NUM_REPEAT: 1
  SOURCE: "svhn"
  VALID_SPLIT_RATIO: 0.5

SOLVER:
  MIN_EPOCHS: 0
  MAX_EPOCHS: 3

OUTPUT:
  PB_FRESH: None
```

Related configuration settings are grouped together. The group headings and allowed values are stored in a separate Python file `config.py` which many users will not need to refer to. The headings and parameters in this example are explained below:

The tutorial YAML file `tutorial.yaml` above overrides certain defaults in `config.py` to make the machine learning process faster and clearer for demonstration purposes.

## 5.3 Customization for your applications

Application of an example to your data can be as simple as creating a new YAML file to (change the defaults to) specify your data location, and other preferred configuration customization, e.g., in the choice of models and/or the number of iterations.

CHAPTER

SIX

# LOAD DATA

## 6.1 Submodules

## 6.2 kale.loaddata.dataset_access module

Dataset Access API adapted from https://github.com/criteo-research/pytorch-ada/blob/master/adalib/ada/datasets/dataset_access.py

**class** kale.loaddata.dataset_access.**DatasetAccess**(*n_classes*)

Bases: object

This class ensures a unique API is used to access training, validation and test splits of any dataset.

**Parameters**
**n_classes** (*int*) – the number of classes.

**n_classes**()

**get_train**()

**Returns: a torch.utils.data.Dataset**
Dataset: a torch.utils.data.Dataset

**get_train_valid**(*valid_ratio*)

Randomly split a dataset into non-overlapping training and validation datasets.

**Parameters**
**valid_ratio** (*float*) – the ratio for validation set

**Returns**
a torch.utils.data.Dataset

**Return type**
Dataset

**get_test**()

kale.loaddata.dataset_access.**get_class_subset**(*dataset*, *class_ids*)

**Parameters**

- **dataset** – a torch.utils.data.Dataset
- **class_ids** (*list, optional*) – List of chosen subset of class ids.

**Returns: a torch.utils.data.Dataset**
Dataset: a torch.utils.data.Dataset with only classes in class_ids

kale.loaddata.dataset_access.**split_by_ratios**(*dataset*, *split_ratios*)

Randomly split a dataset into non-overlapping new datasets of given ratios.

**Parameters**

- **dataset** (*torch.utils.data.Dataset*) – Dataset to be split.
- **split_ratios** (*list*) – Ratios of splits to be produced, where 0 < sum(split_ratios) <= 1.

**Returns**
A list of subsets.

**Return type**
[List]

### Examples

```
>>> import torch
>>> from kale.loaddata.dataset_access import split_by_ratios
>>> subset1, subset2 = split_by_ratios(range(10), [0.3, 0.7])
>>> len(subset1)
3
>>> len(subset2)
7
>>> subset1, subset2 = split_by_ratios(range(10), [0.3])
>>> len(subset1)
3
>>> len(subset2)
7
>>> subset1, subset2, subset3 = split_by_ratios(range(10), [0.3, 0.3])
>>> len(subset1)
3
>>> len(subset2)
3
>>> len(subset3)
4
```

## 6.3 kale.loaddata.image_access

## 6.4 kale.loaddata.mnistm module

Dataset setting and data loader for MNIST-M, from https://github.com/criteo-research/pytorch-ada/blob/master/adalib/ada/datasets/dataset_mnistm.py (based on https://github.com/pytorch/vision/blob/master/torchvision/datasets/mnist.py) CREDIT: https://github.com/corenel

**class** kale.loaddata.mnistm.**MNISTM**(*root*, *train=True*, *transform=None*, *target_transform=None*, *download=False*)

Bases: Dataset

MNIST-M Dataset. Auto-downloads the dataset and provide the torch Dataset API.

**Parameters**

- **root** (*str*) – path to directory where the MNISTM folder will be created (or exists.)

- **train** (*bool, optional*) – defaults to True. If True, loads the training data. Otherwise, loads the test data.
- **transform** (*callable, optional*) – defaults to None. A function/transform that takes in an PIL image and returns a transformed version. E.g., `transforms.RandomCrop` This preprocessing function applied to all images (whether source or target)
- **target_transform** (*callable, optional*) – default to None, similar to transform. This preprocessing function applied to all target images, after *transform*
- **download** (*bool optional*) – defaults to False. Whether to allow downloading the data if not found on disk.

**url = 'https://github.com/VanushVaswani/keras_mnistm/releases/download/1.0/keras_mnistm.pkl.gz'**

**raw_folder = 'raw'**

**processed_folder = 'processed'**

**training_file = 'mnist_m_train.pt'**

**test_file = 'mnist_m_test.pt'**

**download**()

Download the MNISTM data.

## 6.5 kale.loaddata.multi_domain module

Construct a dataset with (multiple) source and target domains, adapted from https://github.com/criteo-research/pytorch-ada/blob/master/adalib/ada/datasets/multisource.py

**class** kale.loaddata.multi_domain.**WeightingType**(*value*)

Bases: Enum

An enumeration.

**NATURAL = 'natural'**

**BALANCED = 'balanced'**

**PRESET0 = 'preset0'**

**class** kale.loaddata.multi_domain.**DatasetSizeType**(*value*)

Bases: Enum

An enumeration.

**Max = 'max'**

**Source = 'source'**

**static get_size**(*size_type*, *source_dataset*, **other_datasets*)

**class** kale.loaddata.multi_domain.**DomainsDatasetBase**

Bases: object

**prepare_data_loaders**()

handles train/validation/test split to have 3 datasets each with data from all domains

**get_domain_loaders**(*split='train'*, *batch_size=32*)

handles the sampling of a dataset containing multiple domains

**Parameters**

- **split** (*string, optional*) – ["train"|"valid"|"test"]. Which dataset to iterate on. Defaults to "train".
- **batch_size** (*int, optional*) – Defaults to 32.

**Returns**

A dataloader with API similar to the torch.dataloader, but returning batches from several domains at each iteration.

**Return type**

*MultiDataLoader*

**class** kale.loaddata.multi_domain.**MultiDomainDatasets**(*source_access:* DatasetAccess, *target_access:* DatasetAccess, *config_weight_type='natural'*, *config_size_type=DatasetSizeType.Max*, *valid_split_ratio=0.1*, *source_sampling_config=None*, *target_sampling_config=None*, *n_fewshot=None*, *random_state=None*, *class_ids=None*)

Bases: *DomainsDatasetBase*

**is_semi_supervised**()

**prepare_data_loaders**()

**get_domain_loaders**(*split='train'*, *batch_size=32*)

**class** kale.loaddata.multi_domain.**MultiDomainImageFolder**(*root: str, loader: ~typing.Callable[[str], ~typing.Any] = <function default_loader>, extensions: ~typing.Optional[~typing.Tuple[str, ...]] = ('.jpg', '.jpeg', '.png', '.ppm', '.bmp', '.pgm', '.tif', '.tiff', '.webp'), transform: ~typing.Optional[~typing.Callable] = None, target_transform: ~typing.Optional[~typing.Callable] = None, sub_domain_set=None, sub_class_set=None, is_valid_file: ~typing.Optional[~typing.Callable[[str], bool]] = None, return_domain_label: ~typing.Optional[bool] = False, split_train_test: ~typing.Optional[bool] = False, split_ratio: float = 0.8*)

Bases: VisionDataset

A generic data loader where the samples are arranged in this way:

```
root/domain_a/class_1/xxx.ext
root/domain_a/class_1/xxy.ext
root/domain_a/class_2/xxz.ext

root/domain_b/class_1/efg.ext
root/domain_b/class_2/pqr.ext
root/domain_b/class_2/lmn.ext

root/domain_k/class_2/123.ext
root/domain_k/class_1/abc3.ext
root/domain_k/class_1/asd932_.ext
```

**Parameters**

- **root** (*string*) – Root directory path.
- **loader** (*callable*) – A function to load a sample given its path.
- **extensions** (*tuple[string]*) – A list of allowed extensions. Either extensions or is_valid_file should be passed.
- **transform** (*callable, optional*) – A function/transform that takes in a sample and returns a transformed version. E.g, `transforms.RandomCrop` for images.
- **target_transform** (*callable, optional*) – A function/transform that takes in the target and transforms it.
- **sub_domain_set** (*list*) – A list of domain names, which should be a subset of domains (folders) under the root directory. If None, all available domains will be used. Defaults to None.
- **sub_class_set** (*list*) – A list of class names, which should be a subset of classes (folders) under each domain's directory. If None, all available classes will be used. Defaults to None.
- **is_valid_file** – A function that takes path of a file and check if the file is a valid file (to check corrupt files). Either extensions or is_valid_file should be passed.

**get_train**()

**get_test**()

kale.loaddata.multi_domain.**make_multi_domain_set**(*directory: str*, *class_to_idx: Dict[str, int]*, *domain_to_idx: Dict[str, int]*, *extensions: Optional[Tuple[str, ...]] = None*, *is_valid_file: Optional[Callable[[str], bool]] = None*) → List[Tuple[str, int, int]]

Generates a list of samples of a form (path_to_sample, class, domain). :param directory: root dataset directory :type directory: str :param class_to_idx: dictionary mapping class name to class index :type class_to_idx: Dict[str, int] :param domain_to_idx: dictionary mapping d name to class index :type domain_to_idx: Dict[str, int] :param extensions: A list of allowed extensions. Either extensions or is_valid_file should be passed.

Defaults to None.

**Parameters**

**is_valid_file** (*optional*) – A function that takes path of a file and checks if the file is a valid file (to check corrupt files) both extensions and is_valid_file should not be passed. Defaults to None.

**Raises**
**ValueError** – In case `extensions` and `is_valid_file` are None or both are not None.

**Returns**
samples of a form (path_to_sample, class, domain)

**Return type**
List[Tuple[str, int, int]]

class kale.loaddata.multi_domain.**ConcatMultiDomainAccess**(*data_access: dict*, *domain_to_idx: dict*, *return_domain_label: Optional[bool] = False*)

Bases: `Dataset`

Concatenate multiple datasets as a single dataset with domain labels

**Parameters**

- **data_access** (*dict*) – Dictionary of domain datasets, e.g. {"Domain1_name": domain1_set, "Domain2_name": domain2_set}
- **domain_to_idx** (*dict*) – Dictionary of domain name to domain labels, e.g. {"Domain1_name": 0, "Domain2_name": 1}
- **return_domain_label** (*Optional[bool], optional*) – Whether return domain labels in each batch. Defaults to False.

class kale.loaddata.multi_domain.**MultiDomainAccess**(*data_access: dict*, *n_classes: int*, *return_domain_label: Optional[bool] = False*)

Bases: *DatasetAccess*

Convert multiple digits-like data accesses to a single data access. :param data_access: Dictionary of data accesses, e.g. {"Domain1_name": domain1_access,

"Domain2_name": domain2_access}

**Parameters**

- **n_classes** (*int*) – number of classes.
- **return_domain_label** (*Optional[bool], optional*) – Whether return domain labels in each batch. Defaults to False.

**get_train**()

**get_test**()

class kale.loaddata.multi_domain.**MultiDomainAdapDataset**(*data_access*, *valid_split_ratio=0.1*, *test_split_ratio=0.2*, *random_state: int = 1*, *test_on_all=False*)

Bases: *DomainsDatasetBase*

The class controlling how the multiple domains are iterated over.

**Parameters**

- **data_access** (`MultiDomainImageFolder, or MultiDomainAccess`) – Multi-domain data access.
- **valid_split_ratio** (*float, optional*) – Split ratio for validation set. Defaults to 0.1.
- **test_split_ratio** (*float, optional*) – Split ratio for test set. Defaults to 0.2.

- **random_state** (*int, optional*) – Random state for generator. Defaults to 1.
- **test_on_all** (*bool, optional*) – Whether test model on all target. Defaults to False.

**prepare_data_loaders**()

**get_domain_loaders**(*split='train'*, *batch_size=32*)

## 6.6 kale.loaddata.polypharmacy_datasets module

**class** kale.loaddata.polypharmacy_datasets.**PolypharmacyDataset**(*url: str*, *root: str*, *name: str*, *mode: str = 'train'*)

Bases: Dataset

Polypharmacy side effect prediction dataset. Only for full-batch training.

**Parameters**

- **url** (*string*) – The url to download the dataset from.
- **root** (*string*) – The root directory containing the dataset file.
- **name** (*string*) – Name of the dataset.
- **mode** (*string*) – "train", "valid" or "test". Defaults to "train".

**load_data**() → Data

Setup dataset: download if need and load it.

## 6.7 kale.loaddata.sampler module

Various sampling strategies for datasets to construct dataloader, from https://github.com/criteo-research/pytorch-ada/blob/master/adalib/ada/datasets/sampler.py

**class** kale.loaddata.sampler.**SamplingConfig**(*balance=False*, *class_weights=None*, *balance_domain=False*)

Bases: object

**create_loader**(*dataset*, *batch_size*)

Create the data loader

Reference: https://pytorch.org/docs/stable/data.html#torch.utils.data.Sampler

**Parameters**

- **dataset** (*Dataset*) – dataset from which to load the data.
- **batch_size** (*int*) – how many samples per batch to load

**class** kale.loaddata.sampler.**FixedSeedSamplingConfig**(*seed=1*, *balance=False*, *class_weights=None*, *balance_domain=False*)

Bases: *SamplingConfig*

**create_loader**(*dataset*, *batch_size*)

Create the data loader with fixed seed.

class kale.loaddata.sampler.**MultiDataLoader**(*dataloaders*, *n_batches*)

Bases: object

Batch Sampler for a MultiDataset. Iterates in parallel over different batch samplers for each dataset. Yields batches [(x_1, y_1), …, (x_s, y_s)] for s datasets.

class kale.loaddata.sampler.**BalancedBatchSampler**(*dataset*, *batch_size*)

Bases: BatchSampler

BatchSampler - from a MNIST-like dataset, samples n_samples for each of the n_classes. Returns batches of size n_classes * (batch_size // n_classes) adapted from https://github.com/adambielski/siamese-triplet/blob/master/datasets.py

class kale.loaddata.sampler.**ReweightedBatchSampler**(*dataset*, *batch_size*, *class_weights*)

Bases: BatchSampler

BatchSampler - from a MNIST-like dataset, samples batch_size according to given input distribution assuming multi-class labels adapted from https://github.com/adambielski/siamese-triplet/blob/master/datasets.py

kale.loaddata.sampler.**get_labels**(*dataset*)

Get class labels for dataset

class kale.loaddata.sampler.**InfiniteSliceIterator**(*array*, *class_*)

Bases: object

**reset**()

**get**(*n*)

class kale.loaddata.sampler.**DomainBalancedBatchSampler**(*dataset*, *batch_size*)

Bases: *BalancedBatchSampler*

**BatchSampler - samples n_samples for each of the n_domains.**
Returns batches of size n_domains * (batch_size / n_domains)

**Parameters**

- **dataset** (*.multi_domain.MultiDomainImageFolder or torch.utils.data.Subset*) – Multi-domain data access.
- **batch_size** (*int*) – Batch size

## 6.8 kale.loaddata.tdc_datasets module

class kale.loaddata.tdc_datasets.**BindingDBDataset**(*name: str*, *split='train'*, *path='./data'*, *mode='cnn_cnn'*, *y_log=True*, *drug_transform=None*, *protein_transform=None*)

Bases: Dataset

A custom dataset for loading and processing original TDC data, which is used as input data in DeepDTA model.

**Parameters**

- **name** (*str*) – TDC dataset name.
- **split** (*str*) – Data split type (train, valid or test).
- **path** (*str*) – dataset download/local load path (default: "./data")

- **mode** (*str*) – encoding mode (default: cnn_cnn)
- **drug_transform** – Transform operation (default: None)
- **protein_transform** – Transform operation (default: None)
- **y_log** (*bool*) – Whether convert y values to log space. (default: True)

## 6.9 kale.loaddata.usps module

Dataset setting and data loader for USPS, from https://github.com/criteo-research/pytorch-ada/blob/master/adalib/ada/datasets/dataset_usps.py (based on https://github.com/mingyuliutw/CoGAN/blob/master/cogan_pytorch/src/dataset_usps.py)

**class** kale.loaddata.usps.**USPS**(*root*, *train=True*, *transform=None*, *download=False*)

Bases: Dataset

USPS Dataset.

**Parameters**

- **root** (*string*) – Root directory of dataset where dataset file exist.
- **train** (*bool, optional*) – If True, resample from dataset randomly.
- **download** (*bool, optional*) – If true, downloads the dataset from the internet and puts it in root directory. If dataset is already downloaded, it is not downloaded again.
- **transform** (*callable, optional*) – A function/transform that takes in an PIL image and returns a transformed version. E.g, transforms.RandomCrop

**url = 'https://raw.githubusercontent.com/mingyuliutw/CoGAN/master/cogan_pytorch/data/uspssample/usps_28x28.pkl'**

**download**()

Download dataset.

**load_samples**()

Load sample images from dataset.

## 6.10 kale.loaddata.video_access module

Action video dataset loading for EPIC-Kitchen, ADL, GTEA, KITCHEN. The code is based on https://github.com/criteo-research/pytorch-ada/blob/master/adalib/ada/datasets/digits_dataset_access.py

kale.loaddata.video_access.**get_image_modality**(*image_modality*)

Change image_modality (string) to rgb (bool) and flow (bool) for efficiency

kale.loaddata.video_access.**get_videodata_config**(*cfg*)

Get the configure parameters for video data from the cfg files

kale.loaddata.video_access.**generate_list**(*data_name*, *data_params_local*, *domain*)

**Parameters**

- **data_name** (*string*) – name of dataset
- **data_params_local** (*dict*) – hyperparameters from configure file

- **domain** (*string*) – domain type (source or target)

**Returns**
image directory of dataset train_listpath (string): training list file directory of dataset test_listpath (string): test list file directory of dataset

**Return type**
data_path (string)

class kale.loaddata.video_access.**VideoDataset**(*value*)

Bases: Enum

An enumeration.

**EPIC = 'EPIC'**

**ADL = 'ADL'**

**GTEA = 'GTEA'**

**KITCHEN = 'KITCHEN'**

static **get_source_target**(*source:* VideoDataset, *target:* VideoDataset, *seed*, *params*)

Gets data loaders for source and target datasets Sets channel_number as 3 for RGB, 2 for flow. Sets class_number as 8 for EPIC, 7 for ADL, 6 for both GTEA and KITCHEN.

**Parameters**

- **source** (VideoDataset) – source dataset name
- **target** (VideoDataset) – target dataset name
- **seed** (*int*) – seed value set manually
- **params** (*CfgNode*) – hyperparameters from configure file

### Examples

```
>>> source, target, num_classes = get_source_target(source, target, seed,␣
↪params)
```

class kale.loaddata.video_access.**VideoDatasetAccess**(*data_path*, *train_list*, *test_list*, *image_modality*, *frames_per_segment*, *n_classes*, *transform_kind*, *seed*)

Bases: *DatasetAccess*

Common API for video dataset access

**Parameters**

- **data_path** (*string*) – image directory of dataset
- **train_list** (*string*) – training list file directory of dataset
- **test_list** (*string*) – test list file directory of dataset
- **image_modality** (*string*) – image type (RGB or Optical Flow)
- **frames_per_segment** (*int*) – length of each action sample (the unit is number of frame)
- **n_classes** (*int*) – number of class
- **transform_kind** (*string*) – types of video transforms

- **seed** (*int*) – seed value set manually

**get_train_valid**(*valid_ratio*)

Get the train and validation dataset with the fixed random split. This is used for joint input like RGB and optical flow, which will call *get_train_valid* twice. Fixing the random seed here can keep the seeds for twice the same.

**class** kale.loaddata.video_access.**EPICDatasetAccess**(*data_path*, *train_list*, *test_list*, *image_modality*, *frames_per_segment*, *n_classes*, *transform_kind*, *seed*)

Bases: *VideoDatasetAccess*

EPIC data loader

**get_train**()

**get_test**()

**class** kale.loaddata.video_access.**GTEADatasetAccess**(*data_path*, *train_list*, *test_list*, *image_modality*, *frames_per_segment*, *n_classes*, *transform_kind*, *seed*)

Bases: *VideoDatasetAccess*

GTEA data loader

**get_train**()

**get_test**()

**class** kale.loaddata.video_access.**ADLDatasetAccess**(*data_path*, *train_list*, *test_list*, *image_modality*, *frames_per_segment*, *n_classes*, *transform_kind*, *seed*)

Bases: *VideoDatasetAccess*

ADL data loader

**get_train**()

**get_test**()

**class** kale.loaddata.video_access.**KITCHENDatasetAccess**(*data_path*, *train_list*, *test_list*, *image_modality*, *frames_per_segment*, *n_classes*, *transform_kind*, *seed*)

Bases: *VideoDatasetAccess*

KITCHEN data loader

**get_train**()

**get_test**()

## 6.11 kale.loaddata.video_datasets module

class kale.loaddata.video_datasets.**BasicVideoDataset**(*root_path: str*, *annotationfile_path: str*, *dataset_split: str*, *image_modality: str*, *num_segments: int = 1*, *frames_per_segment: int = 16*, *imagefile_template: str = 'img_{:010d}.jpg'*, *transform=None*, *random_shift: bool = True*, *test_mode: bool = False*, *n_classes: int = 8*)

Bases: *VideoFrameDataset*

Dataset for GTEA, ADL and KITCHEN.

**Parameters**

- **root_path** (*string*) – The root path in which video folders lie.
- **annotationfile_path** (*string*) – The annotation file containing one row per video sample.
- **dataset_split** (*string*) – Split type (train or test)
- **image_modality** (*string*) – Image modality (RGB or Optical Flow)
- **num_segments** (*int*) – The number of segments the video should be divided into to sample frames from.
- **frames_per_segment** (*int*) – The number of frames that should be loaded per segment.
- **imagefile_template** (*string*) – The image filename template.
- **transform** (*Compose*) – Video transform.
- **random_shift** (*bool*) – Whether the frames from each segment should be taken consecutively starting from the center(False) of the segment, or consecutively starting from a random(True) location inside the segment range.
- **test_mode** (*bool*) – Whether this is a test dataset. If so, chooses frames from segments with random_shift=False.
- **n_classes** (*int*) – The number of classes.

**make_dataset**()

Load data from the EPIC-Kitchen list file and make them into the united format. Different datasets correspond to a different number of classes.

**Returns**
list of (video_name, start_frame, end_frame, label)

**Return type**
data (list)

class kale.loaddata.video_datasets.**EPIC**(*root_path: str*, *annotationfile_path: str*, *dataset_split: str*, *image_modality: str*, *num_segments: int = 1*, *frames_per_segment: int = 16*, *imagefile_template: str = 'img_{:010d}.jpg'*, *transform=None*, *random_shift: bool = True*, *test_mode: bool = False*, *n_classes: int = 8*)

Bases: *VideoFrameDataset*

Dataset for EPIC-Kitchen.

**make_dataset**()

Load data from the EPIC-Kitchen list file and make them into the united format. Because the original list files are not the same, inherit from class BasicVideoDataset and be modified.

## 6.12 kale.loaddata.video_multi_domain module

Construct a dataset for videos with (multiple) source and target domains

**class** kale.loaddata.video_multi_domain.**VideoMultiDomainDatasets**(*source_access_dict*, *target_access_dict*, *image_modality*, *seed*, *config_weight_type='natural'*, *config_size_type=DatasetSizeType.Max*, *valid_split_ratio=0.1*, *source_sampling_config=None*, *target_sampling_config=None*, *n_fewshot=None*, *random_state=None*, *class_ids=None*)

Bases: *MultiDomainDatasets*

**prepare_data_loaders**()

**get_domain_loaders**(*split='train'*, *batch_size=32*)

## 6.13 kale.loaddata.videos module

**class** kale.loaddata.videos.**VideoFrameDataset**(*root_path: str*, *annotationfile_path: str*, *image_modality: str = 'rgb'*, *num_segments: int = 3*, *frames_per_segment: int = 1*, *imagefile_template: str = 'img_{:05d}.jpg'*, *transform=None*, *random_shift: bool = True*, *test_mode: bool = False*)

Bases: `Dataset`

A highly efficient and adaptable dataset class for videos. Instead of loading every frame of a video, loads x RGB frames of a video (sparse temporal sampling) and evenly chooses those frames from start to end of the video, returning a list of x PIL images or `FRAMES x CHANNELS x HEIGHT x WIDTH` tensors where FRAMES=x if the `kale.prepdata.video_transform.ImglistToTensor()` transform is used.

More specifically, the frame range [START_FRAME, END_FRAME] is divided into NUM_SEGMENTS segments and FRAMES_PER_SEGMENT consecutive frames are taken from each segment.

---

**Note:** A demonstration of using this class can be seen in `PyKale/examples/video_loading` https://github.com/pykale/pykale/tree/master/examples/video_loading

---

---

**Note:** This dataset broadly corresponds to the frame sampling technique introduced in `Temporal Segment Networks` at ECCV2016 https://arxiv.org/abs/1608.00859.

---

**Note:** This class relies on receiving video data in a structure where inside a `ROOT_DATA` folder, each video lies in its own folder, where each video folder contains the frames of the video as individual files with a naming convention such as img_001.jpg … img_059.jpg. For enumeration and annotations, this class expects to receive the path to a .txt file where each video sample has a row with four (or more in the case of multi-label, see example README on Github) space separated values: `VIDEO_FOLDER_PATH START_FRAME END_FRAME LABEL_INDEX`. `VIDEO_FOLDER_PATH` is expected to be the path of a video folder excluding the `ROOT_DATA` prefix. For example, `ROOT_DATA` might be `home\data\datasetxyz\videos\`, inside of which a `VIDEO_FOLDER_PATH` might be `jumping\0052\` or `sample1\` or `00053\`.

**Parameters**

- **root_path** – The root path in which video folders lie. this is ROOT_DATA from the description above.
- **annotationfile_path** – The .txt annotation file containing one row per video sample as described above.
- **image_modality** – Image modality (RGB or Optical Flow).
- **num_segments** – The number of segments the video should be divided into to sample frames from.
- **frames_per_segment** – The number of frames that should be loaded per segment. For each segment's frame-range, a random start index or the center is chosen, from which frames_per_segment consecutive frames are loaded.
- **imagefile_template** – The image filename template that video frame files have inside of their video folders as described above.
- **transform** – Transform pipeline that receives a list of PIL images/frames.
- **random_shift** – Whether the frames from each segment should be taken consecutively starting from the center of the segment, or consecutively starting from a random location inside the segment range.
- **test_mode** – Whether this is a test dataset. If so, chooses frames from segments with random_shift=False.

## 6.14 Module contents

CHAPTER

SEVEN

# PREPROCESS DATA

## 7.1 Submodules

## 7.2 kale.prepdata.chem_transform module

Functions for labeling and encoding chemical characters like Compound SMILES and atom string, refer to https://github.com/hkmztrk/DeepDTA and https://github.com/thinng/GraphDTA.

kale.prepdata.chem_transform.**integer_label_smiles**(*smiles*, *max_length=85*, *isomeric=False*)

Integer encoding for SMILES string sequence.

**Parameters**

- **smiles** (*str*) – Simplified molecular-input line-entry system, which is a specification in the form of a line
- **strings.** (*notation for describing the structure of chemical species using short ASCII*) –
- **max_length** (*int*) – Maximum encoding length of input SMILES string. (default: 85)
- **isomeric** (*bool*) – Whether the input SMILES string includes isomeric information (default: False).

kale.prepdata.chem_transform.**integer_label_protein**(*sequence*, *max_length=1200*)

Integer encoding for protein string sequence.

**Parameters**

- **sequence** (*str*) – Protein string sequence.
- **max_length** – Maximum encoding length of input protein string. (default: 1200)

## 7.3 kale.prepdata.graph_negative_sampling module

kale.prepdata.graph_negative_sampling.**negative_sampling**(*pos_edge_index: Tensor*, *num_nodes: int*) → Tensor

Negative sampling for link prediction. Copy-paste from https://github.com/NYXFLOWER/GripNet.

**Parameters**

- **pos_edge_index** (*torch.Tensor*) – edge indices in COO format with shape [2, num_edges].

- **num_nodes** (*int*) – the number of nodes in the graph.

**Returns**
edge indices in COO format with shape [2, num_edges].

**Return type**
torch.Tensor

kale.prepdata.graph_negative_sampling.**typed_negative_sampling**(*pos_edge_index: Tensor*, *num_nodes: int*, *range_list: Tensor*) → Tensor

Typed negative sampling for link prediction. Copy-paste from https://github.com/NYXFLOWER/GripNet.

**Parameters**

- **pos_edge_index** (*torch.Tensor*) – edge indices in COO format with shape [2, num_edges].
- **num_nodes** (*int*) – the number of nodes in the graph.
- **range_list** (*torch.Tensor*) – the range of edge types. [[start_index, end_index], ...]

**Returns**
edge indices in COO format with shape [2, num_edges].

**Return type**
torch.Tensor

## 7.4 kale.prepdata.image_transform module

## 7.5 kale.prepdata.supergraph_construct module

## 7.6 kale.prepdata.tensor_reshape module

kale.prepdata.tensor_reshape.**spatial_to_seq**(*image_tensor: Tensor*)

Takes a torch tensor of shape (batch_size, channels, height, width) as used and outputted by CNNs and creates a sequence view of shape (sequence_length, batch_size, channels) as required by torch's transformer module. In other words, unrolls the spatial grid into the sequence length and rearranges the dimension ordering.

**Parameters**
**image_tensor** – tensor of shape (batch_size, channels, height, width) (required).

kale.prepdata.tensor_reshape.**seq_to_spatial**(*sequence_tensor: Tensor*, *desired_height: int*, *desired_width: int*)

Takes a torch tensor of shape (sequence_length, batch_size, num_features) as used and outputted by Transformers and creates a view of shape (batch_size, num_features, height, width) as used and outputted by CNNs. In other words, rearranges the dimension ordering and rolls sequence_length into (height,width). height*width must equal the sequence length of the input sequence.

**Parameters**

- **sequence_tensor** – sequence tensor of shape (sequence_length, batch_size, num_features) (required).
- **desired_height** – the height into which the sequence length should be rolled into (required).

- **desired_width** – the width into which the sequence length should be rolled into (required).

## 7.7 kale.prepdata.video_transform module

kale.prepdata.video_transform.**get_transform**(*kind*, *image_modality*)

Define transforms (for commonly used datasets)

**Parameters**

- **kind** (*[type]*) – the dataset (transformation) name
- **image_modality** (*string*) – image type (RGB or Optical Flow)

**class** kale.prepdata.video_transform.**ImglistToTensor**

Bases: Module

Converts a list of PIL images in the range [0,255] to a torch.FloatTensor of shape (NUM_IMAGES x CHANNELS x HEIGHT x WIDTH) in the range [0,1]. Can be used as first transform for kale.loaddata.videos.VideoFrameDataset.

**forward**(*img_list*)

For RGB input, converts each PIL image in a list to a torch Tensor and stacks them into a single tensor. For flow input, converts every two PIL images (x(u)_img, y(v)_img) in a list to a torch Tensor and stacks them. For example, if input list size is 16, the dimension is [16, 1, 224, 224] and the frame order is [frame 1_x, frame 1_y, frame 2_x, frame 2_y, frame 3_x, ..., frame 8_x, frame 8_y]. The output will be [[frame 1_x, frame 1_y], [frame 2_x, frame 2_y], [frame 3_x, ..., [frame 8_x, frame 8_y]] and the dimension is [8, 2, 224, 224].

**Parameters**
**img_list** – list of PIL images.

**Returns**
tensor of size `` NUM_IMAGES x CHANNELS x HEIGHT x WIDTH``

**class** kale.prepdata.video_transform.**TensorPermute**

Bases: Module

Convert a torch.FloatTensor of shape (NUM_IMAGES x CHANNELS x HEIGHT x WIDTH) to a torch.FloatTensor of shape (CHANNELS x NUM_IMAGES x HEIGHT x WIDTH).

## 7.8 Module contents

CHAPTER

# EIGHT

# EMBED

## 8.1 Submodules

## 8.2 kale.embed.attention_cnn module

**class** kale.embed.attention_cnn.**ContextCNNGeneric**(*cnn: Module*, *cnn_output_shape: Tuple[int, int, int, int]*, *contextualizer: Union[Module, Any]*, *output_type: str*)

Bases: `Module`

A template to construct a feature extractor consisting of a CNN followed by a sequence-to-sequence contextualizer like a Transformer-Encoder. Before inputting the CNN output tensor to the contextualizer, the tensor's spatial dimensions are unrolled into a sequence.

**Parameters**

- **cnn** (*nn.Module*) – any convolutional neural network that takes in batches of images of shape (batch_size, channels, height, width) and outputs tensor representations of shape (batch_size, out_channels, out_height, out_width).
- **cnn_output_shape** (*tuple*) – A tuple of shape (batch_size, num_channels, height, width) describing the output shape of the given CNN (required).
- **contextualizer** (*nn.Module, optional*) – A sequence-to-sequence model that takes inputs of shape (num_timesteps, batch_size, num_features) and uses attention to contextualize the sequence and returns a sequence of the exact same shape. This will mainly be a Transformer-Encoder (required).
- **output_type** (*string*) – One of 'sequence' or 'spatial'. If Spatial then the final output of the model, which is a sequence, will be reshaped to resemble the image-batch shape of the output of the CNN. If Sequence then the output sequence is returned as is (required).

### Examples

```
>>> cnn = nn.Sequential(nn.Conv2d(3, 32, kernel_size=3),
>>>                     nn.Conv2d(32, 64, kernel_size=3),
>>>                     nn.MaxPool2d(2))
>>> cnn_output_shape = (-1, 64, 8, 8)
>>> contextualizer = nn.TransformerEncoderLayer(...)
>>> output_type = 'spatial'
>>>
```

(continues on next page)

(continued from previous page)

```
>>> attention_cnn = ContextCNNGeneric(cnn, cnn_output_shape, contextualizer, output_
↪type)
>>> output = attention_cnn(torch.randn((32,3,16,16)))
>>>
>>> output.size() == cnn_output_shape # True
```

**forward**(*x: Tensor*)

Pass the input through the cnn and then the contextualizer.

**Parameters**

**x** – input image batch exactly as for CNNs (required).

**training: bool**

**class** kale.embed.attention_cnn.**CNNTransformer**(*cnn: Module*, *cnn_output_shape: Tuple[int, int, int, int]*, *num_layers: int*, *num_heads: int*, *dim_feedforward: int*, *dropout: float*, *output_type: str*, *positional_encoder: Optional[Module] = None*)

Bases: *ContextCNNGeneric*

A feature extractor consisting of a given CNN backbone followed by a standard Transformer-Encoder. See documentation of "ContextCNNGeneric" for more information.

**Parameters**

- **cnn** (*nn.Module*) – any convolutional neural network that takes in batches of images of shape (batch_size, channels, height, width) and outputs tensor representations of shape (batch_size, out_channels, out_height, out_width) (required).
- **cnn_output_shape** (*tuple*) – a tuple of shape (batch_size, num_channels, height, width) describing the output shape of the given CNN (required).
- **num_layers** (*int*) – number of attention layers in the Transformer-Encoder (required).
- **num_heads** (*int*) – number of attention heads in each transformer block (required).
- **dim_feedforward** (*int*) – number of neurons in the intermediate dense layer of each transformer feedforward block (required).
- **dropout** (*float*) – dropout rate of the transformer layers (required).
- **output_type** (*string*) – one of 'sequence' or 'spatial'. If Spatial then the final output of the model, which is the sequence output of the Transformer-Encoder, will be reshaped to resemble the image-batch shape of the output of the CNN (required).
- **positional_encoder** (*nn.Module*) – None or a nn.Module that expects inputs of shape (sequence_length, batch_size, embedding_dim) and returns the same input after adding some positional information to the embeddings. If *None*, then the default and fixed sin-cos positional encodings of base transformers are applied (optional).

### Examples

See pykale/examples/cifar_cnntransformer/model.py

**training: bool**

## 8.3 kale.embed.factorization module

Python implementation of a tensor factorization algorithm Multilinear Principal Component Analysis (MPCA) and a matrix factorization algorithm Maximum Independence Domain Adaptation (MIDA

**class** kale.embed.factorization.**MPCA**(*var_ratio=0.97*, *max_iter=1*, *vectorize=False*, *n_components=None*)

Bases: BaseEstimator, TransformerMixin

MPCA implementation compatible with sickit-learn

**Parameters**

- **var_ratio** (*float, optional*) – Percentage of variance explained (between 0 and 1). Defaults to 0.97.
- **max_iter** (*int, optional*) – Maximum number of iteration. Defaults to 1.
- **vectorize** (*bool*) – Whether return the transformed/projected tensor in vector. Defaults to False.
- **n_components** (*int*) – Number of components to keep. Applies only when vectorize=True. Defaults to None.

**proj_mats**

A list of transposed projection matrices, shapes (P_1, I_1), ..., (P_N, I_N), where P_1, ..., P_N are output tensor shape for each sample.

**Type**
list of arrays

**idx_order**

The ordering index of projected (and vectorized) features in decreasing variance.

**Type**
array-like

**mean_**

Per-feature empirical mean, estimated from the training set, shape (I_1, I_2, ..., I_N).

**Type**
array-like

**shape_in**

Input tensor shapes, i.e. (I_1, I_2, ..., I_N).

**Type**
tuple

**shape_out**

Output tensor shapes, i.e. (P_1, P_2, ..., P_N).

**Type**
tuple

**Reference:**

Haiping Lu, K.N. Plataniotis, and A.N. Venetsanopoulos, "MPCA: Multilinear Principal Component Analysis of Tensor Objects", IEEE Transactions on Neural Networks, Vol. 19, No. 1, Page: 18-39, January 2008. For initial Matlab implementation, please go to https://uk.mathworks.com/matlabcentral/fileexchange/26168.

### Examples

```
>>> import numpy as np
>>> from kale.embed.factorization import MPCA
>>> x = np.random.random((40, 20, 25, 20))
>>> x.shape
(40, 20, 25, 20)
>>> mpca = MPCA()
>>> x_projected = mpca.fit_transform(x)
>>> x_projected.shape
(40, 18, 23, 18)
>>> x_projected = mpca.transform(x)
>>> x_projected.shape
(40, 7452)
>>> x_projected = mpca.transform(x)
>>> x_projected.shape
(40, 50)
>>> x_rec = mpca.inverse_transform(x_projected)
>>> x_rec.shape
(40, 20, 25, 20)
```

**fit**(*x*, *y=None*)

Fit the model with input training data x.

**Args**

**x (array-like tensor): Input data, shape (n_samples, I_1, I_2, …, I_N), where n_samples is the number of**
samples, I_1, I_2, …, I_N are the dimensions of corresponding mode (1, 2, …, N), respectively.

y (None): Ignored variable.

**Returns**

self (object). Returns the instance itself.

**transform**(*x*)

Perform dimension reduction on x

**Parameters**

**x** (*array-like tensor*) – Data to perform dimension reduction, shape (n_samples, I_1, I_2, …, I_N).

**Returns**

Projected data in lower dimension, shape (n_samples, P_1, P_2, …, P_N) if self.vectorize==False. If self.vectorize==True, features will be sorted based on their explained variance ratio, shape (n_samples, P_1 * P_2 * … * P_N) if self.n_components is None, and shape (n_samples, n_components) if self.n_component is a valid integer.

**Return type**

array-like tensor

`inverse_transform`(*x*)

Reconstruct projected data to the original shape and add the estimated mean back

**Parameters**
**x** (*array-like tensor*) – Data to be reconstructed, shape (n_samples, P_1, P_2, …, P_N), if self.vectorize == False, where P_1, P_2, …, P_N are the reduced dimensions of corresponding mode (1, 2, …, N), respectively. If self.vectorize == True, shape (n_samples, self.n_components) or shape (n_samples, P_1 * P_2 * … * P_N).

**Returns**
Reconstructed tensor in original shape, shape (n_samples, I_1, I_2, …, I_N)

**Return type**
array-like tensor

`class kale.embed.factorization.MIDA`(*n_components*, *kernel='linear'*, *lambda_=1.0*, *mu=1.0*, *eta=1.0*, *augmentation=False*, *kernel_params=None*)

Bases: `BaseEstimator`, `TransformerMixin`

Maximum independence domain adaptation :param n_components: Number of components to keep. :type n_components: int :param kernel: "linear", "rbf", or "poly". Kernel to use for MIDA. Defaults to "linear". :type kernel: str :param mu: Hyperparameter of the l2 penalty. Defaults to 1.0. :type mu: float :param eta: Hyperparameter of the label dependence. Defaults to 1.0. :type eta: float :param augmentation: Whether using covariates as augment features. Defaults to False. :type augmentation: bool :param kernel_params: Parameters for the kernel. Defaults to None. :type kernel_params: dict or None

### References

**[1] Yan, K., Kou, L. and Zhang, D., 2018. Learning domain-invariant subspace using domain features and**
independence maximization. IEEE transactions on cybernetics, 48(1), pp.288-299.

`fit`(*x*, *y=None*, *covariates=None*)

**Parameters**

- **x** – array-like. Input data, shape (n_samples, n_features)
- **y** – array-like. Labels, shape (nl_samples,)
- **covariates** – array-like. Domain co-variates, shape (n_samples, n_co-variates)

---

**Note:** Unsupervised MIDA is performed if y is None. Semi-supervised MIDA is performed is y is not None.

---

`fit_transform`(*x*, *y=None*, *covariates=None*)

**Parameters**

- **x** – array-like, shape (n_samples, n_features)
- **y** – array-like, shape (n_samples,)
- **covariates** – array-like, shape (n_samples, n_covariates)

**Returns**
array-like, shape (n_samples, n_components)

**Return type**
x_transformed

`transform`(*x*, *covariates=None*)

**Parameters**

- **x** – array-like, shape (n_samples, n_features)
- **covariates** – array-like, augmentation features, shape (n_samples, n_covariates)

**Returns**
array-like, shape (n_samples, n_components)

**Return type**
x_transformed

## 8.4 kale.embed.gcn module

`class kale.embed.gcn.GCNEncoderLayer`(*in_channels*, *out_channels*, *improved=False*, *cached=False*, *bias=True*, ***kwargs*)

Bases: `MessagePassing`

Modification of PyTorch Geometirc's nn.GCNConv, which reduces the computational cost of GCN layer for GripNet model. The graph convolutional operator from the "Semi-supervised Classification with Graph Convolutional Networks" (ICLR 2017) paper.

$$\mathbf{X}' = \hat{\mathbf{D}}^{-1/2}\hat{\mathbf{A}}\hat{\mathbf{D}}^{-1/2}\mathbf{X}\mathbf{\Theta},$$

where $\hat{\mathbf{A}} = \mathbf{A} + \mathbf{I}$ denotes the adjacency matrix with inserted self-loops and $\hat{D}_{ii} = \sum_{j=0} \hat{A}_{ij}$ its diagonal degree matrix.

Note: For more information please see Pytorch Geomertic's nn.GCNConv docs.

**Parameters**

- **in_channels** (*int*) – Size of each input sample.
- **out_channels** (*int*) – Size of each output sample.
- **improved** (*bool, optional*) – If set to `True`, the layer computes $\hat{\mathbf{A}}$ as $\mathbf{A} + 2\mathbf{I}$. (default: `False`)
- **cached** (*bool, optional*) – If set to `True`, the layer will cache the computation of $\hat{\mathbf{D}}^{-1/2}\hat{\mathbf{A}}\hat{\mathbf{D}}^{-1/2}$ on first execution, and will use the cached version for further executions. This parameter should only be set to `True` in transductive learning scenarios. (default: `False`)
- **bias** (*bool, optional*) – If set to `False`, the layer will not learn an additive bias. (default: `True`)
- ****kwargs** (*optional*) – Additional arguments of `torch_geometric.nn.conv.MessagePassing`.

`reset_parameters`()

`static norm`(*edge_index*, *num_nodes*, *edge_weight*, *improved=False*, *dtype=None*)

Add self-loops and apply symmetric normalization

**forward**(*x*, *edge_index*, *edge_weight=None*)

**Parameters**

- **x** (*torch.Tensor*) – The input node feature embedding.
- **edge_index** (*torch.Tensor*) – Graph edge index in COO format with shape [2, num_edges].
- **edge_weight** (*torch.Tensor, optional*) – The one-dimensional relation weight for each edge in `edge_index` (default: None).

class kale.embed.gcn.**RGCNEncoderLayer**(*in_channels*, *out_channels*, *num_relations*, *num_bases*, *after_relu*, *bias=False*, ***kwargs*)

Bases: `MessagePassing`

Modification of PyTorch Geometirc's nn.RGCNConv, which reduces the computational and memory cost of RGCN encoder layer for GripNet model. The relational graph convolutional operator from the "Modeling Relational Data with Graph Convolutional Networks" paper.

$$\mathbf{x}_i' = \mathbf{\Theta}_{\text{root}} \cdot \mathbf{x}_i + \sum_{r \in \mathcal{R}} \sum_{j \in \mathcal{N}_r(i)} \frac{1}{|\mathcal{N}_r(i)|} \mathbf{\Theta}_r \cdot \mathbf{x}_j,$$

where $\mathcal{R}$ denotes the set of relations, *i.e.* edge types. Edge type needs to be a one-dimensional `torch.long` tensor which stores a relation identifier $\in \{0, \ldots, |\mathcal{R}| - 1\}$ for each edge.

Note: For more information please see Pytorch Geomertic's nn.RGCNConv docs.

**Parameters**

- **in_channels** (*int*) – Size of each input sample.
- **out_channels** (*int*) – Size of each output sample.
- **num_relations** (*int*) – Number of edge relations.
- **num_bases** (*int*) – Use bases-decoposition regulatization scheme and num_bases denotes the number of bases.
- **after_relu** (*bool*) – Whether input embedding is activated by relu function or not.
- **bias** (*bool*) – If set to `False`, the layer will not learn an additive bias. (default: `False`)
- ****kwargs** (*optional*) – Additional arguments of `torch_geometric.nn.conv.MessagePassing`.

**reset_parameters**()

**forward**(*x*, *edge_index*, *edge_type*, *range_list*)

**Parameters**

- **x** (*torch.Tensor*) – The input node feature embedding.
- **edge_index** (*torch.Tensor*) – Graph edge index in COO format with shape [2, num_edges].
- **edge_type** (*torch.Tensor*) – The one-dimensional relation type/index for each edge in `edge_index`.
- **range_list** (*torch.Tensor*) – The index range list of each edge type with shape [num_types, 2].

## 8.5 kale.embed.gripnet module

## 8.6 kale.embed.image_cnn module

CNNs for extracting features from small images of size 32x32 (e.g. MNIST) and regular images of size 224x224 (e.g. ImageNet). The code is based on https://github.com/criteo-research/pytorch-ada/blob/master/adalib/ada/models/modules.py,

which is for domain adaptation.

**class** kale.embed.image_cnn.**SmallCNNFeature**(*num_channels=3*, *kernel_size=5*)

Bases: Module

A feature extractor for small 32x32 images (e.g. CIFAR, MNIST) that outputs a feature vector of length 128.

**Parameters**

- **num_channels** (*int*) – the number of input channels (default=3).
- **kernel_size** (*int*) – the size of the convolution kernel (default=5).

**Examples::**

```
>>> feature_network = SmallCNNFeature(num_channels)
```

**forward**(*input_*)

**output_size**()

**training: bool**

**class** kale.embed.image_cnn.**SimpleCNNBuilder**(*conv_layers_spec*, *activation_fun='relu'*, *use_batchnorm=True*, *pool_locations=(0, 3)*, *num_channels=3*)

Bases: Module

A builder for simple CNNs to experiment with different basic architectures.

**Parameters**

- **num_channels** (*int*) – the number of input channels. Defaults to 3.
- **conv_layers_spec** (*list*) – a list for each convolutional layer given as [num_channels, kernel_size]. For example, [[16, 3], [16, 1]] represents 2 layers with 16 filters and kernel sizes of 3 and 1 respectively.
- **activation_fun** (*str*) – a string specifying the activation function to use. one of ('relu', 'elu', 'leaky_relu'). Defaults to "relu".
- **use_batchnorm** (*boolean*) – a boolean flag indicating whether to use batch normalization. Defaults to True.
- **pool_locations** (*tuple*) – the index after which pooling layers should be placed in the convolutional layer list. Defaults to (0,3). (0,3) means placing 2 pooling layers after the first and fourth convolutional layer.
- **num_channels** – the number of input channels. Defaults to 3.

activations = {'elu': ELU(alpha=1.0), 'leaky_relu': LeakyReLU(negative_slope=0.01), 'relu': ReLU()}

forward(*x*)

training: bool

class kale.embed.image_cnn.**ResNet18Feature**(*pretrained=True*)

Bases: Module

Modified ResNet18 (without the last layer) feature extractor for regular 224x224 images.

**Parameters**
**pretrained** (*bool*) – If True, returns a model pre-trained on ImageNet

**Note:** Code adapted by pytorch-ada from https://github.com/thuml/Xlearn/blob/master/pytorch/src/network.py

forward(*x*)

output_size()

training: bool

class kale.embed.image_cnn.**ResNet34Feature**(*pretrained=True*)

Bases: Module

Modified ResNet34 (without the last layer) feature extractor for regular 224x224 images.

**Parameters**
**pretrained** (*bool*) – If True, returns a model pre-trained on ImageNet

**Note:** Code adapted by pytorch-ada from https://github.com/thuml/Xlearn/blob/master/pytorch/src/network.py

forward(*x*)

output_size()

training: bool

class kale.embed.image_cnn.**ResNet50Feature**(*pretrained=True*)

Bases: Module

Modified ResNet50 (without the last layer) feature extractor for regular 224x224 images.

**Parameters**
**pretrained** (*bool*) – If True, returns a model pre-trained on ImageNet

**Note:** Code adapted by pytorch-ada from https://github.com/thuml/Xlearn/blob/master/pytorch/src/network.py

forward(*x*)

output_size()

training: bool

class kale.embed.image_cnn.**ResNet101Feature**(*pretrained=True*)

Bases: Module

Modified ResNet101 (without the last layer) feature extractor for regular 224x224 images.

**Parameters**
**pretrained** (*bool*) – If True, returns a model pre-trained on ImageNet

**Note:** Code adapted by pytorch-ada from https://github.com/thuml/Xlearn/blob/master/pytorch/src/network.py

**forward**(*x*)

**output_size**()

**training: bool**

class kale.embed.image_cnn.**ResNet152Feature**(*pretrained=True*)

Bases: Module

Modified ResNet152 (without the last layer) feature extractor for regular 224x224 images.

**Parameters**
**pretrained** (*bool*) – If True, returns a model pre-trained on ImageNet

**Note:** Code adapted by pytorch-ada from https://github.com/thuml/Xlearn/blob/master/pytorch/src/network.py

**forward**(*x*)

**output_size**()

**training: bool**

## 8.7 kale.embed.positional_encoding module

class kale.embed.positional_encoding.**PositionalEncoding**(*d_model: int*, *max_len: int = 5000*)

Bases: Module

Implements the positional encoding as described in the NIPS2017 paper 'Attention Is All You Need' about Transformers (https://arxiv.org/abs/1706.03762). Essentially, for all timesteps in a given sequence, adds information about the relative temporal location of a timestep directly into the features of that timestep, and then returns this slightly-modified, same-shape sequence.

**Parameters**

- **d_model** – the number of features that each timestep has (required).
- **max_len** – the maximum sequence length that the positional encodings should support (required).

**forward**(*x*)

Expects input of shape (sequence_length, batch_size, num_features) and returns output of the same shape. sequence_length is at most allowed to be self.max_len and num_features is expected to be exactly self.d_model

**Parameters**
**x** – a sequence input of shape (sequence_length, batch_size, num_features) (required).

`training: bool`

## 8.8 kale.embed.seq_nn module

DeepDTA based models for drug-target interaction prediction problem.

`class kale.embed.seq_nn.CNNEncoder(num_embeddings, embedding_dim, sequence_length, num_kernels, kernel_length)`

Bases: `Module`

The DeepDTA's CNN encoder module, which comprises three 1D-convolutional layers and one max-pooling layer. The module is applied to encoding drug/target sequence information, and the input should be transformed information with integer/label encoding. The original paper is "DeepDTA: deep drug–target binding affinity prediction" .

**Parameters**

- **num_embeddings** (*int*) – Number of embedding labels/categories, depends on the types of encoding sequence.
- **embedding_dim** (*int*) – Dimension of embedding labels/categories.
- **sequence_length** (*int*) – Max length of input sequence.
- **num_kernels** (*int*) – Number of kernels (filters).
- **kernel_length** (*int*) – Length of kernel (filter).

`forward(x)`

`training: bool`

`class kale.embed.seq_nn.GCNEncoder(in_channel=78, out_channel=128, dropout_rate=0.2)`

Bases: `Module`

The GraphDTA's GCN encoder module, which comprises three graph convolutional layers and one full connected layer. The model is a variant of DeepDTA and is applied to encoding drug molecule graph information. The original paper is "GraphDTA: Predicting drug–target binding affinity with graph neural networks" .

**Parameters**

- **in_channel** (*int*) – Dimension of each input node feature.
- **out_channel** (*int*) – Dimension of each output node feature.
- **dropout_rate** (*float*) – dropout rate during training.

`forward(x, edge_index, batch)`

`training: bool`

## 8.9 kale.embed.video_feature_extractor module

Define the feature extractor for video including I3D, R3D_18, MC3_18 and R2PLUS1D_18 w/o SELayers.

kale.embed.video_feature_extractor.**get_video_feat_extractor**(*model_name*, *image_modality*, *attention*, *num_classes*)

Get the feature extractor w/o the pre-trained model and SELayers. The pre-trained models are saved in the path `$XDG_CACHE_HOME/torch/hub/checkpoints/`. For Linux, default path is `~/.cache/torch/hub/checkpoints/`. For Windows, default path is `C:/Users/$USER_NAME/.cache/torch/hub/checkpoints/`. Provide four pre-trained models: "rgb_imagenet", "flow_imagenet", "rgb_charades", "flow_charades".

**Parameters**

- **model_name** (*string*) – The name of the feature extractor. (Choices=["I3D", "R3D_18", "R2PLUS1D_18", "MC3_18"])
- **image_modality** (*string*) – Image type. (Choices=["rgb", "flow", "joint"])
- **attention** (*string*) – The attention type. (Choices=["SELayerC", "SELayerT", "SELayerCoC", "SELayerMC", "SELayerCT", "SELayerTC", "SELayerMAC"])
- **num_classes** (*int*) – The class number of specific dataset. (Default: No use)

**Returns**

The network to extract features. class_feature_dim (int): The dimension of the feature network output for ClassNet.

It is a convention when the input dimension and the network is fixed.

domain_feature_dim (int): The dimension of the feature network output for DomainNet.

**Return type**
feature_network (dictionary)

## 8.10 kale.embed.video_i3d module

Define Inflated 3D ConvNets(I3D) on Action Recognition from https://ieeexplore.ieee.org/document/8099985 Created by Xianyuan Liu from modifying https://github.com/piergiaj/pytorch-i3d/blob/master/pytorch_i3d.py and https://github.com/deepmind/kinetics-i3d/blob/master/i3d.py

**class** kale.embed.video_i3d.**MaxPool3dSamePadding**(*kernel_size: Union[int, Tuple[int, ...]]*, *stride: Optional[Union[int, Tuple[int, ...]]] = None*, *padding: Union[int, Tuple[int, ...]] = 0*, *dilation: Union[int, Tuple[int, ...]] = 1*, *return_indices: bool = False*, *ceil_mode: bool = False*)

Bases: `MaxPool3d`

Construct 3d max pool with same padding. PyTorch does not provide same padding. Same padding means the output size matches input size for stride=1.

**compute_pad**(*dim*, *s*)

Get the zero padding number.

**forward**(*x*)

Compute 'same' padding. Add zero to the back position first.

**kernel_size: Union[int, Tuple[int, int, int]]**

**stride: Union[int, Tuple[int, int, int]]**

**padding: Union[int, Tuple[int, int, int]]**

**dilation: Union[int, Tuple[int, int, int]]**

class kale.embed.video_i3d.**Unit3D**(*in_channels*, *output_channels*, *kernel_shape=(1, 1, 1)*, *stride=(1, 1, 1)*, *padding=0*, *activation_fn=<function relu>*, *use_batch_norm=True*, *use_bias=False*, *name='unit_3d'*)

Bases: Module

Basic unit containing Conv3D + BatchNorm + non-linearity.

**compute_pad**(*dim*, *s*)

Get the zero padding number.

**forward**(*x*)

Connects the module to inputs. Dynamically pad based on input size in forward function. :param x: Inputs to the Unit3D component.

**Returns**
Outputs from the module.

**training: bool**

class kale.embed.video_i3d.**InceptionModule**(*in_channels*, *out_channels*, *name*)

Bases: Module

Construct Inception module. Concatenation after four branches (1x1x1 conv; 1x1x1 + 3x3x3 convs; 1x1x1 + 3x3x3 convs; 3x3x3 max-pool + 1x1x1 conv). In *forward*, we check if SELayers are used, which are channel-wise (SELayerC), temporal-wise (SELayerT), channel-temporal-wise (SELayerTC & SELayerCT).

**forward**(*x*)

**training: bool**

class kale.embed.video_i3d.**InceptionI3d**(*num_classes=400*, *spatial_squeeze=True*, *final_endpoint='Logits'*, *name='inception_i3d'*, *in_channels=3*, *dropout_keep_prob=0.5*)

Bases: Module

Inception-v1 I3D architecture. The model is introduced in:

Quo Vadis, Action Recognition? A New Model and the Kinetics Dataset Joao Carreira, Andrew Zisserman https://arxiv.org/pdf/1705.07750v1.pdf.

**See also the Inception architecture, introduced in:**
Going deeper with convolutions Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, Andrew Rabinovich. http://arxiv.org/pdf/1409.4842v1.pdf.

**VALID_ENDPOINTS = ('Conv3d_1a_7x7', 'MaxPool3d_2a_3x3', 'Conv3d_2b_1x1', 'Conv3d_2c_3x3', 'MaxPool3d_3a_3x3', 'Mixed_3b', 'Mixed_3c', 'MaxPool3d_4a_3x3', 'Mixed_4b', 'Mixed_4c', 'Mixed_4d', 'Mixed_4e', 'Mixed_4f', 'MaxPool3d_5a_2x2', 'Mixed_5b', 'Mixed_5c', 'Logits', 'Predictions')**

**replace_logits**(*num_classes*)

Update the output size with num_classes according to the specific setting.

**build**()

**forward**(*x*)

The output is the result of the final average pooling layer with 1024 dimensions.

**extract_features**(*x*)

**training: bool**

kale.embed.video_i3d.**i3d**(*name*, *num_channels*, *num_classes*, *pretrained=False*, *progress=True*)

Get InceptionI3d module w/o pretrained model.

kale.embed.video_i3d.**i3d_joint**(*rgb_pt*, *flow_pt*, *num_classes*, *pretrained=False*, *progress=True*)

Get I3D models for different inputs.

**Parameters**

- **rgb_pt** (*string, optional*) – the name of pre-trained model for RGB input.
- **flow_pt** (*string, optional*) – the name of pre-trained model for flow input.
- **num_classes** (*int*) – the class number of dataset.
- **pretrained** (*bool*) – choose if pretrained parameters are used. (Default: False)
- **progress** (*bool, optional*) – whether or not to display a progress bar to stderr. (Default: True)

**Returns**

A dictionary contains RGB and flow models.

**Return type**

models (dictionary)

## 8.11 kale.embed.video_res3d module

Define MC3_18, R3D_18, R2plus1D_18 on Action Recognition from https://arxiv.org/abs/1711.11248 Created by Xianyuan Liu from modifying https://github.com/pytorch/vision/blob/master/torchvision/models/video/resnet.py

**class** kale.embed.video_res3d.**Conv3DSimple**(*in_planes*, *out_planes*, *midplanes=None*, *stride=1*, *padding=1*)

Bases: Conv3d

3D convolutions for R3D (3x3x3 kernel)

**static get_downsample_stride**(*stride*)

**bias: Optional[Tensor]**

**out_channels: int**

**kernel_size: Tuple[int, ...]**

**stride: Tuple[int, ...]**

**padding: Union[str, Tuple[int, ...]]**

**dilation: Tuple[int, ...]**

**transposed: bool**

**output_padding: Tuple[int, ...]**

**groups: int**

**padding_mode: str**

**weight: Tensor**

**class** kale.embed.video_res3d.**Conv2Plus1D**(*in_planes*, *out_planes*, *midplanes*, *stride=1*, *padding=1*)

Bases: Sequential

(2+1)D convolutions for R2plus1D (1x3x3 kernel + 3x1x1 kernel)

**static get_downsample_stride**(*stride*)

**training: bool**

**class** kale.embed.video_res3d.**Conv3DNoTemporal**(*in_planes*, *out_planes*, *midplanes=None*, *stride=1*, *padding=1*)

Bases: Conv3d

3D convolutions without temporal dimension for MCx (1x3x3 kernel)

**static get_downsample_stride**(*stride*)

**bias: Optional[Tensor]**

**out_channels: int**

**kernel_size: Tuple[int, ...]**

**stride: Tuple[int, ...]**

**padding: Union[str, Tuple[int, ...]]**

**dilation: Tuple[int, ...]**

**transposed: bool**

**output_padding: Tuple[int, ...]**

**groups: int**

**padding_mode: str**

**weight: Tensor**

**class** kale.embed.video_res3d.**BasicBlock**(*inplanes*, *planes*, *conv_builder*, *stride=1*, *downsample=None*)

Bases: Module

Basic ResNet building block. Each block consists of two convolutional layers with a ReLU activation function after each layer and residual connections. In *forward*, we check if SELayers are used, which are channel-wise (SELayerC) and temporal-wise (SELayerT).

**expansion = 1**

**forward**(*x*)

**training: bool**

class kale.embed.video_res3d.**Bottleneck**(*inplanes*, *planes*, *conv_builder*, *stride=1*, *downsample=None*)

Bases: Module

BottleNeck building block. Default: No use. Each block consists of two 1*n*n and one n*n*n convolutional layers with a ReLU activation function after each layer and residual connections.

**expansion = 4**

**forward**(*x*)

**training: bool**

class kale.embed.video_res3d.**BasicStem**

Bases: Sequential

The default conv-batchnorm-relu stem. The first layer normally. (64 3x7x7 kernels)

**training: bool**

class kale.embed.video_res3d.**BasicFLowStem**

Bases: Sequential

The default stem for optical flow.

**training: bool**

class kale.embed.video_res3d.**R2Plus1dStem**

Bases: Sequential

R(2+1)D stem is different than the default one as it uses separated 3D convolution. (45 1x7x7 kernels + 64 3x1x1 kernel)

**training: bool**

class kale.embed.video_res3d.**R2Plus1dFlowStem**

Bases: Sequential

R(2+1)D stem for optical flow.

**training: bool**

class kale.embed.video_res3d.**VideoResNet**(*block*, *conv_makers*, *layers*, *stem*, *num_classes=400*, *zero_init_residual=False*)

Bases: Module

**replace_fc**(*num_classes*, *block=<class 'kale.embed.video_res3d.BasicBlock'>*)

Update the output size with num_classes according to the specific setting.

**forward**(*x*)

**training: bool**

kale.embed.video_res3d.**r3d_18_rgb**(*pretrained=False*, *progress=True*, ***kwargs*)

Construct 18 layer Resnet3D model for RGB as in https://arxiv.org/abs/1711.11248

**Parameters**

- **pretrained** (*bool*) – If True, returns a model pre-trained on Kinetics-400
- **progress** (*bool*) – If True, displays a progress bar of the download to stderr

**Returns**

R3D-18 network

**Return type**
nn.Module

kale.embed.video_res3d.**r3d_18_flow**(*pretrained=False*, *progress=True*, ***kwargs*)
Construct 18 layer Resnet3D model for optical flow.

kale.embed.video_res3d.**mc3_18_rgb**(*pretrained=False*, *progress=True*, ***kwargs*)
Constructor for 18 layer Mixed Convolution network for RGB as in https://arxiv.org/abs/1711.11248

**Parameters**

- **pretrained** (*bool*) – If True, returns a model pre-trained on Kinetics-400
- **progress** (*bool*) – If True, displays a progress bar of the download to stderr

**Returns**
MC3 Network definition

**Return type**
nn.Module

kale.embed.video_res3d.**mc3_18_flow**(*pretrained=False*, *progress=True*, ***kwargs*)
Constructor for 18 layer Mixed Convolution network for optical flow.

kale.embed.video_res3d.**r2plus1d_18_rgb**(*pretrained=False*, *progress=True*, ***kwargs*)
Constructor for the 18 layer deep R(2+1)D network for RGB as in https://arxiv.org/abs/1711.11248

**Parameters**

- **pretrained** (*bool*) – If True, returns a model pre-trained on Kinetics-400
- **progress** (*bool*) – If True, displays a progress bar of the download to stderr

**Returns**
R(2+1)D-18 network

**Return type**
nn.Module

kale.embed.video_res3d.**r2plus1d_18_flow**(*pretrained=False*, *progress=True*, ***kwargs*)
Constructor for the 18 layer deep R(2+1)D network for optical flow.

kale.embed.video_res3d.**r3d**(*rgb=False*, *flow=False*, *pretrained=False*, *progress=True*)
Get R3D_18 models.

kale.embed.video_res3d.**mc3**(*rgb=False*, *flow=False*, *pretrained=False*, *progress=True*)
Get MC3_18 models.

kale.embed.video_res3d.**r2plus1d**(*rgb=False*, *flow=False*, *pretrained=False*, *progress=True*)
Get R2PLUS1D_18 models.

## 8.12 kale.embed.video_se_i3d module

Add SELayers to I3D

class kale.embed.video_se_i3d.**SEInceptionI3DRGB**(*num_channels*, *num_classes*, *attention*)

Bases: Module

Add the several SELayers to I3D for RGB input. :param num_channels: the channel number of the input. :type num_channels: int :param num_classes: the class number of dataset. :type num_classes: int :param attention: the name of the SELayer.

(Options: ["SELayerC", "SELayerT", "SELayerCoC", "SELayerMC", "SELayerMAC", "SELayerCT" and "SELayerTC"])

**Returns**
I3D model with SELayers.

**Return type**
model (*VideoResNet*)

**forward**(*x*)

**training: bool**

class kale.embed.video_se_i3d.**SEInceptionI3DFlow**(*num_channels*, *num_classes*, *attention*)

Bases: Module

Add the several SELayers to I3D for optical flow input.

**forward**(*x*)

**training: bool**

kale.embed.video_se_i3d.**se_inception_i3d**(*name*, *num_channels*, *num_classes*, *attention*, *pretrained=False*, *progress=True*, *rgb=True*)

Get InceptionI3d module w/o SELayer and pretrained model.

kale.embed.video_se_i3d.**se_i3d_joint**(*rgb_pt*, *flow_pt*, *num_classes*, *attention*, *pretrained=False*, *progress=True*)

Get I3D models with SELayers for different inputs.

**Parameters**

- **rgb_pt** (*string, optional*) – the name of pre-trained model for RGB input.
- **flow_pt** (*string, optional*) – the name of pre-trained model for optical flow input.
- **num_classes** (*int*) – the class number of dataset.
- **attention** (*string, optional*) – the name of the SELayer.
- **pretrained** (*bool*) – choose if pretrained parameters are used. (Default: False)
- **progress** (*bool, optional*) – whether or not to display a progress bar to stderr. (Default: True)

**Returns**
A dictionary contains models for RGB and optical flow.

**Return type**
models (dictionary)

## 8.13 kale.embed.video_se_res3d module

Add SELayers to MC3_18, R3D_18, R2plus1D_18

kale.embed.video_se_res3d.**se_r3d_18_rgb**(*attention*, *pretrained=False*, *progress=True*, ***kwargs*)

kale.embed.video_se_res3d.**se_r3d_18_flow**(*attention*, *pretrained=False*, *progress=True*, ***kwargs*)

kale.embed.video_se_res3d.**se_mc3_18_rgb**(*attention*, *pretrained=False*, *progress=True*, ***kwargs*)

kale.embed.video_se_res3d.**se_mc3_18_flow**(*attention*, *pretrained=False*, *progress=True*, ***kwargs*)

kale.embed.video_se_res3d.**se_r2plus1d_18_rgb**(*attention*, *pretrained=False*, *progress=True*, ***kwargs*)

kale.embed.video_se_res3d.**se_r2plus1d_18_flow**(*attention*, *pretrained=False*, *progress=True*, ***kwargs*)

kale.embed.video_se_res3d.**se_r3d**(*attention*, *rgb=False*, *flow=False*, *pretrained=False*, *progress=True*)

Get R3D_18 models with SELayers for different inputs.

**Parameters**

- **attention** (*string*) – the name of the SELayer.
- **rgb** (*bool*) – choose if RGB model is needed. (Default: False)
- **flow** (*bool*) – choose if optical flow model is needed. (Default: False)
- **pretrained** (*bool*) – choose if pretrained parameters are used. (Default: False)
- **progress** (*bool, optional*) – whether or not to display a progress bar to stderr. (Default: True)

**Returns**

A dictionary contains models for RGB and optical flow.

**Return type**

models (dictionary)

kale.embed.video_se_res3d.**se_mc3**(*attention*, *rgb=False*, *flow=False*, *pretrained=False*, *progress=True*)

Get MC3_18 models with SELayers for different inputs.

kale.embed.video_se_res3d.**se_r2plus1d**(*attention*, *rgb=False*, *flow=False*, *pretrained=False*, *progress=True*)

Get R2+1D_18 models with SELayers for different inputs.

## 8.14 kale.embed.video_selayer module

Python implementation of Squeeze-and-Excitation Layers (SELayer) Initial implementation: channel-wise (SELayerC) Improved implementation: temporal-wise (SELayerT), convolution-based channel-wise (SELayerCoC), max-pooling-based channel-wise (SELayerMC), multi-pooling-based channel-wise (SELayerMAC)

[Redundancy and repeat of code will be reduced in the future.]

### References

Hu Jie, Li Shen, and Gang Sun. "Squeeze-and-excitation networks." In CVPR, pp. 7132-7141. 2018. For initial implementation, please go to https://github.com/hujie-frank/SENet

kale.embed.video_selayer.**get_selayer**(*attention*)

Get SELayers referring to attention.

**Parameters**
**attention** (*string*) – the name of the SELayer. (Options: ["SELayerC", "SELayerT", "SELayerCoC", "SELayerMC", "SELayerMAC"])

**Returns**
the SELayer.

**Return type**
se_layer (*SELayer*, optional)

**class** kale.embed.video_selayer.**SELayer**(*channel*, *reduction=16*)

Bases: Module

Helper class for SELayer design.

**forward**(*x*)

**training: bool**

**class** kale.embed.video_selayer.**SELayerC**(*channel*, *reduction=16*)

Bases: *SELayer*

Construct channel-wise SELayer.

**forward**(*x*)

**training: bool**

**class** kale.embed.video_selayer.**SELayerT**(*channel*, *reduction=2*)

Bases: *SELayer*

Construct temporal-wise SELayer.

**forward**(*x*)

**training: bool**

**class** kale.embed.video_selayer.**SELayerCoC**(*channel*, *reduction=16*)

Bases: *SELayer*

Construct convolution-based channel-wise SELayer.

**forward**(*x*)

**training: bool**

**class** kale.embed.video_selayer.**SELayerMC**(*channel*, *reduction=16*)

Bases: *SELayer*

Construct channel-wise SELayer with max pooling.

**forward**(*x*)

**training: bool**

**class** kale.embed.video_selayer.**SELayerMAC**(*channel*, *reduction=16*)

Bases: *SELayer*

Construct channel-wise SELayer with the mix of average pooling and max pooling.

**forward**(*x*)

**training: bool**

## 8.15 Module contents

CHAPTER

NINE

# PREDICT

## 9.1 Submodules

## 9.2 kale.predict.class_domain_nets module

Classification of data or domain

Modules for typical classification tasks (into class labels) and adversarial discrimination of source vs target domains, from https://github.com/criteo-research/pytorch-ada/blob/master/adalib/ada/models/modules.py

**class** kale.predict.class_domain_nets.**SoftmaxNet**(*input_dim=15*, *n_classes=2*, *name='c'*, *hidden=()*, *activation_fn=<class 'torch.nn.modules.activation.ReLU'>*, ***activation_args*)

Bases: Module

Regular and domain classifier network for regular-size images

**Parameters**

- **input_dim** (*int, optional*) – the dimension of the final feature vector.. Defaults to 15.
- **n_classes** (*int, optional*) – the number of classes. Defaults to 2.
- **name** (*str, optional*) – the classifier name. Defaults to "c".
- **hidden** (*tuple, optional*) – the hidden layer sizes. Defaults to ().
- **activation_fn** (*[type], optional*) – the activation function. Defaults to nn.ReLU.

**forward**(*input_data*)

**extra_repr**()

**n_classes**()

**training: bool**

**class** kale.predict.class_domain_nets.**ClassNet**(*n_class=10*, *input_shape=(-1, 64, 8, 8)*)

Bases: Module

Simple classification prediction-head block to plug ontop of the 4D output of a CNN.

**Parameters**

- **n_class** (*int, optional*) – the number of different classes that can be predicted. Defaults to 10.

- **input_shape** (*tuples, optional*) – the shape that input to this head will have. Expected to be (batch_size, channels, height, width). Defaults to (-1, 64, 8, 8).

**forward**(*x*)

**training: bool**

class kale.predict.class_domain_nets.**ClassNetSmallImage**(*input_size=128*, *n_class=10*)

Bases: Module

Regular classifier network for small-size images

**Parameters**

- **input_size** (*int, optional*) – the dimension of the final feature vector. Defaults to 128.
- **n_class** (*int, optional*) – the number of classes. Defaults to 10.

**n_classes**()

**forward**(*input*)

**training: bool**

class kale.predict.class_domain_nets.**DomainNetSmallImage**(*input_size=128*, *bigger_discrim=False*)

Bases: Module

Domain classifier network for small-size images

**Parameters**

- **input_size** (*int, optional*) – the dimension of the final feature vector. Defaults to 128.
- **bigger_discrim** (*bool, optional*) – whether to use deeper network. Defaults to False.

**forward**(*input*)

**training: bool**

class kale.predict.class_domain_nets.**ClassNetVideo**(*input_size=512*, *n_channel=100*, *dropout_keep_prob=0.5*, *n_class=8*)

Bases: Module

Regular classifier network for video input.

**Parameters**

- **input_size** (*int, optional*) – the dimension of the final feature vector. Defaults to 512.
- **n_channel** (*int, optional*) – the number of channel for Linear and BN layers.
- **dropout_keep_prob** (*int, optional*) – the dropout probability for keeping the parameters.
- **n_class** (*int, optional*) – the number of classes. Defaults to 8.

**n_classes**()

**forward**(*input*)

**training: bool**

class kale.predict.class_domain_nets.**ClassNetVideoConv**(*input_size=1024*, *n_class=8*)

Bases: Module

Classifier network for video input refer to MMSADA.

**Parameters**

- **input_size** (*int, optional*) – the dimension of the final feature vector. Defaults to 1024.
- **n_class** (*int, optional*) – the number of classes. Defaults to 8.

### References

Munro Jonathan, and Dima Damen. "Multi-modal domain adaptation for fine-grained action recognition." In CVPR, pp. 122-132. 2020.

**forward**(*input*)

**training: bool**

class kale.predict.class_domain_nets.**DomainNetVideo**(*input_size=128*, *n_channel=100*)

Bases: Module

Regular domain classifier network for video input.

**Parameters**

- **input_size** (*int, optional*) – the dimension of the final feature vector. Defaults to 512.
- **n_channel** (*int, optional*) – the number of channel for Linear and BN layers.

**forward**(*input*)

**training: bool**

## 9.3 kale.predict.decode module

## 9.4 kale.predict.isonet module

The ISONet module, which is based on the ResNet module, from https://github.com/HaozhiQi/ISONet/blob/master/isonet/models/isonet.py (based on https://github.com/facebookresearch/pycls/blob/master/pycls/models/resnet.py)

kale.predict.isonet.**get_trans_fun**(*name*)

Retrieves the transformation function by name.

class kale.predict.isonet.**SReLU**(*nc*)

Bases: Module

Shifted ReLU

**forward**(*x*)

**training: bool**

class kale.predict.isonet.**ResHead**(*w_in*, *net_params*)

Bases: Module

ResNet head.

**forward**(*x*)

**training: bool**

class kale.predict.isonet.**BasicTransform**(*w_in*, *w_out*, *stride*, *has_bn*, *use_srelu*, *w_b=None*, *num_gs=1*)

Bases: Module

Basic transformation: 3x3, 3x3

**forward**(*x*)

**training: bool**

class kale.predict.isonet.**BottleneckTransform**(*w_in*, *w_out*, *stride*, *has_bn*, *use_srelu*, *w_b*, *num_gs*)

Bases: Module

Bottleneck transformation: 1x1, 3x3, 1x1, only for very deep networks

**forward**(*x*)

**training: bool**

class kale.predict.isonet.**ResBlock**(*w_in*, *w_out*, *stride*, *trans_fun*, *has_bn*, *has_st*, *use_srelu*, *w_b=None*, *num_gs=1*)

Bases: Module

Residual block: x + F(x)

**forward**(*x*)

**training: bool**

class kale.predict.isonet.**ResStage**(*w_in*, *w_out*, *stride*, *net_params*, *d*, *w_b=None*, *num_gs=1*)

Bases: Module

Stage of ResNet.

**forward**(*x*)

**training: bool**

class kale.predict.isonet.**ResStem**(*w_in*, *w_out*, *net_params*, *kernelsize=3*, *stride=1*, *padding=1*, *use_maxpool=False*, *poolksize=3*, *poolstride=2*, *poolpadding=1*)

Bases: Module

Stem of ResNet.

**forward**(*x*)

**training: bool**

class kale.predict.isonet.**ISONet**(*net_params*)

Bases: Module

ISONet, a modified ResNet model.

**forward**(*x*)

**ortho**(*device*)

regularizes the convolution kernel to be (near) orthogonal during training. This is called in Trainer.loss of the isonet example.

**ortho_conv**(*m*, *device*)

regularizes the convolution kernel to be (near) orthogonal during training.

**Parameters**
**m** (*nn.module]*) – [description]

**training: bool**

## 9.5 kale.predict.losses module

Commonly used losses, from domain adaptation package https://github.com/criteo-research/pytorch-ada/blob/master/adalib/ada/models/losses.py

kale.predict.losses.**cross_entropy_logits**(*output*, *target*, *weights=None*)

Computes cross entropy with logits

**Parameters**

- **output** (*Tensor*) – The output of the last layer of the network, before softmax.
- **target** (*Tensor*) – The ground truth label.
- **weights** (*Tensor, optional*) – The weight of each sample. Defaults to None.

### Examples

See DANN, WDGRL, and MMD trainers in kale.pipeline.domain_adapter

kale.predict.losses.**topk_accuracy**(*output*, *target*, *topk=(1,)*)

Computes the top-k accuracy for the specified values of k.

**Parameters**

- **output** (*Tensor*) – output (Tensor): The output of the last layer of the network, before softmax. Shape: (batch_size, class_count).
- **target** (*Tensor*) – The ground truth label. Shape: (batch_size)
- **topk** (*tuple(int)*) – Compute accuracy at top-k for the values of k specified in this parameter.

**Returns**
A list of tensors of the same length as topk. Each tensor consists of boolean variables to show if this prediction ranks top k with each value of k. True means the prediction ranks top k and False means not. The shape of tensor is batch_size, i.e. the number of predictions.

**Return type**
list(Tensor)

### Examples

```
>>> output = torch.tensor(([0.3, 0.2, 0.1], [0.3, 0.2, 0.1]))
>>> target = torch.tensor((0, 1))
>>> top1, top2 = topk_accuracy(output, target, topk=(1, 2)) # get the boolean value
>>> top1_value = top1.double().mean() # get the top 1 accuracy score
>>> top2_value = top2.double().mean() # get the top 2 accuracy score
```

kale.predict.losses.**multitask_topk_accuracy**(*output*, *target*, *topk=(1,)*)

Computes the top-k accuracy for the specified values of k for multitask input.

**Parameters**

- **output** (*tuple(Tensor)*) – A tuple of generated predictions. Each tensor is of shape [batch_size, class_count], class_count can vary per task basis, i.e. outputs[i].shape[1] can differ from outputs[j].shape[1].
- **target** (*tuple(Tensor)*) – A tuple of ground truth. Each tensor is of shape [batch_size]
- **topk** (*tuple(int)*) – Compute accuracy at top-k for the values of k specified in this parameter.

**Returns**

A list of tensors of the same length as topk. Each tensor consists of boolean variables to show if predictions of multitask ranks top k with each value of k. True means predictions of this output for all tasks ranks top k and False means not. The shape of tensor is batch_size, i.e. the number of predictions.

### Examples

```
>>> first_output = torch.tensor(([0.3, 0.2, 0.1], [0.3, 0.2, 0.1]))
>>> first_target = torch.tensor((0, 2))
>>> second_output = torch.tensor(([0.2, 0.1], [0.2, 0.1]))
>>> second_target = torch.tensor((0, 1))
>>> output = (first_output, second_output)
>>> target = (first_target, second_target)
>>> top1, top2 = multitask_topk_accuracy(output, target, topk=(1, 2)) #␣
↪get the boolean value
>>> top1_value = top1.double().mean() # get the top 1 accuracy score
>>> top2_value = top2.double().mean() # get the top 2 accuracy score
```

**Return type**

list(Tensor)

kale.predict.losses.**entropy_logits**(*linear_output*)

Computes entropy logits in CDAN with entropy conditioning (CDAN+E)

### Examples

See CDANTrainer in kale.pipeline.domain_adapter

kale.predict.losses.**entropy_logits_loss**(*linear_output*)

Computes entropy logits loss in semi-supervised or few-shot domain adaptation

### Examples

See FewShotDANNTrainer in kale.pipeline.domain_adapter

kale.predict.losses.**gradient_penalty**(*critic*, *h_s*, *h_t*)

Computes gradient penalty in Wasserstein distance guided representation learning

### Examples

See WDGRLTrainer and WDGRLTrainerMod in kale.pipeline.domain_adapter

kale.predict.losses.**gaussian_kernel**(*source*, *target*, *kernel_mul=2.0*, *kernel_num=5*, *fix_sigma=None*)

Code from XLearn: computes the full kernel matrix, which is less than optimal since we don't use all of it with the linear MMD estimate.

### Examples

See DANTrainer and JANTrainer in kale.pipeline.domain_adapter

kale.predict.losses.**compute_mmd_loss**(*kernel_values*, *batch_size*)

Computes the Maximum Mean Discrepancy (MMD) between domains.

### Examples

See DANTrainer and JANTrainer in kale.pipeline.domain_adapter

kale.predict.losses.**hsic**(*kx*, *ky*, *device*)

Perform independent test with Hilbert-Schmidt Independence Criterion (HSIC) between two sets of variables x and y.

**Parameters**

- **kx** (*2-D tensor*) – kernel matrix of x, shape (n_samples, n_samples)
- **ky** (*2-D tensor*) – kernel matrix of y, shape (n_samples, n_samples)
- **device** (*torch.device*) – the desired device of returned tensor

**Returns**

Independent test score >= 0

**Return type**

[tensor]

**Reference:**

**[1] Gretton, Arthur, Bousquet, Olivier, Smola, Alex, and Schölkopf, Bernhard. Measuring Statistical Dependence**

with Hilbert-Schmidt Norms. In Algorithmic Learning Theory (ALT), pp. 63–77. 2005.

**[2] Gretton, Arthur, Fukumizu, Kenji, Teo, Choon H., Song, Le, Schölkopf, Bernhard, and Smola, Alex J. A Kernel**
Statistical Test of Independence. In Advances in Neural Information Processing Systems, pp. 585–592. 2008.

kale.predict.losses.**euclidean**(*x1*, *x2*)

Compute the Euclidean distance

**Parameters**

- **x1** (*torch.Tensor*) – variables set 1
- **x2** (*torch.Tensor*) – variables set 2

**Returns**
Euclidean distance

**Return type**
torch.Tensor

## 9.6 Module contents

CHAPTER

# TEN

# EVALUATE

## 10.1 Submodules

## 10.2 kale.evaluate.metrics module

kale.evaluate.metrics.**concord_index**(*y*, *y_pred*)

Calculate the Concordance Index (CI), which is a metric to measure the proportion of concordant pairs between real and predict values.

**Parameters**

- **y** (*array*) – real values.
- **y_pred** (*array*) – predicted values.

kale.evaluate.metrics.**auprc_auroc_ap**(*target: Tensor*, *score: Tensor*)

auprc: area under the precision-recall curve auroc: area under the receiver operating characteristic curve ap: average precision

Copy-paste from https://github.com/NYXFLOWER/GripNet

## 10.3 Module contents

CHAPTER

# ELEVEN

# INTERPRET

## 11.1 Submodules

## 11.2 kale.interpret.model_weights module

kale.interpret.model_weights.**select_top_weight**(*weights*, *select_ratio: float = 0.05*)

Select top weights in magnitude, and the rest of weights will be zeros

**Parameters**

- **weights** (*array-like*) – model weights, can be a vector or a higher order tensor
- **select_ratio** (*float, optional*) – ratio of top weights to be selected. Defaults to 0.05.

**Returns**

top weights in the same shape with the input model weights

**Return type**

array-like

## 11.3 kale.interpret.visualize module

kale.interpret.visualize.**plot_weights**(*weight_img*, *background_img=None*, *color_marker_pos='rs'*, *color_marker_neg='gs'*, *im_kwargs=None*, *marker_kwargs=None*)

Visualize model weights

**Parameters**

- **weight_img** (*array-like*) – Model weight/coefficients in 2D, could be a 2D slice of a 3D or higher order tensor.
- **background_img** (*array-like, optional*) – 2D background image. Defaults to None.
- **color_marker_pos** (*str, optional*) – Color and marker for weights in positive values. Defaults to red "rs".
- **color_marker_neg** (*str, optional*) – Color and marker for weights in negative values. Defaults to blue "gs".
- **im_kwargs** (*dict, optional*) – Keyword arguments for background images. Defaults to None.
- **marker_kwargs** (*dict, optional*) – Keyword arguments for background images. Defaults to None.

**Returns**
Figure to plot.

**Return type**
[matplotlib.figure.Figure]

kale.interpret.visualize.**plot_multi_images**(*images*, *n_cols=1*, *n_rows=None*, *marker_locs=None*, *image_titles=None*, *marker_titles=None*, *marker_cmap=None*, *figsize=None*, *im_kwargs=None*, *marker_kwargs=None*, *legend_kwargs=None*, *title_kwargs=None*)

Plot multiple images with markers in one figure.

**Parameters**

- **images** (*array-like*) – Images to plot, shape(n_samples, dim1, dim2)
- **n_cols** (*int, optional*) – Number of columns for plotting multiple images. Defaults to 1.
- **n_rows** (*int, optional*) – Number of rows for plotting multiple images. If None, n_rows = n_samples / n_cols.
- **marker_locs** (*array-like, optional*) – Locations of markers, shape (n_samples, 2 * n_markers). Defaults to None.
- **marker_titles** (*list, optional*) – Names of the markers, where len(marker_names) == n_markers. Defaults to None.
- **marker_cmap** (*str, optional*) – Name of the color map used for plotting markers. Default to None.
- **image_titles** (*list, optional*) – List of title for each image, where len(image_names) == n_samples. Defaults to None.
- **figsize** (*tuple, optional*) – Figure size. Defaults to None.
- **im_kwargs** (*dict, optional*) – Keyword arguments for plotting images. Defaults to None.
- **marker_kwargs** (*dict, optional*) – Keyword arguments for markers. Defaults to None.
- **legend_kwargs** (*dict, optional*) – Keyword arguments for legend. Defaults to None.
- **title_kwargs** (*dict, optional*) – Keyword arguments for title. Defaults to None.

**Returns**
Figure to plot.

**Return type**
[matplotlib.figure.Figure]

kale.interpret.visualize.**distplot_1d**(*data*, *labels=None*, *xlabel=None*, *ylabel=None*, *title=None*, *figsize=None*, *colors=None*, *title_kwargs=None*, *hist_kwargs=None*)

Plot distribution of 1D data.

**Parameters**

- **data** (*array-like or list*) – Data to plot.
- **labels** (*list, optional*) – List of labels for each data. Defaults to None.
- **xlabel** (*str, optional*) – Label for x-axis. Defaults to None.

- **ylabel** (*str, optional*) – Label for y-axis. Defaults to None.
- **title** (*str, optional*) – Title of the plot. Defaults to None.
- **figsize** (*tuple, optional*) – Figure size. Defaults to None.
- **colors** (*str, optional*) – Color of the line. Defaults to None.
- **title_kwargs** (*dict, optional*) – Keyword arguments for title. Defaults to None.
- **hist_kwargs** (*dict, optional*) – Keyword arguments for histogram. Defaults to None.

**Returns**
Figure to plot.

**Return type**
[matplotlib.figure.Figure]

## 11.4 Module contents

CHAPTER

# TWELVE

# PIPELINE

## 12.1 Submodules

## 12.2 kale.pipeline.base_nn_trainer module

Classification systems (pipelines)

This module provides neural network (nn) trainers for developing classification task models. The BaseNNTrainer defines the required fundamental functions and structures, such as the optimizer, learning rate scheduler, training/validation/testing procedure, workflow, etc. The BaseNNTrainer is inherited to construct specialized trainers.

The structure and workflow of BaseNNTrainer is consistent with *kale.pipeline.domain_adapter.BaseAdaptTrainer*

This module uses PyTorch Lightning to standardize the flow.

**class** kale.pipeline.base_nn_trainer.**BaseNNTrainer**(*optimizer*, *max_epochs*, *init_lr=0.001*, *adapt_lr=False*)

Bases: `LightningModule`

Base class for classification models using neural network, based on PyTorch Lightning wrapper. The forward pass and loss computation must be implemented if new trainers inherit from this class. The basic workflow is defined in this class as follows. Every training/validation/testing procedure will call *compute_loss()* to compute the loss and log the output metrics. The *compute_loss()* function will call *forward()* to generate the output feature using the neural networks.

**Parameters**

- **optimizer** (*dict, None*) – optimizer parameters.
- **max_epochs** (*int*) – maximum number of epochs.
- **init_lr** (*float*) – initial learning rate. Defaults to 0.001.
- **adapt_lr** (*bool*) – whether to use the schedule for the learning rate. Defaults to False.

**forward**(*x*)

Override this function to define the forward pass. Normally includes feature extraction and classification and be called in *compute_loss()*.

**compute_loss**(*batch*, *split_name='valid'*)

Compute loss for a given batch.

**Parameters**

- **batch** (*tuple*) – batches returned by dataloader.

- **split_name** (*str, optional*) – learning stage (one of ["train", "valid", "test"]). Defaults to "valid" for validation. "train" is for training and "test" for testing. This is currently used only for naming the metrics used for logging.

**Returns**

loss value. log_metrics (dict): dictionary of metrics to be logged. This is needed when using PyKale logging, but not

mandatory when using PyTorch Lightning logging.

**Return type**
loss (torch.Tensor)

**configure_optimizers**()

Default optimizer configuration. Set Adam to the default and provide SGD with cosine annealing. If other optimizers are needed, please override this function.

**training_step**(*train_batch*, *batch_idx*) → Tensor

Compute and return the training loss and metrics on one step. loss is to store the loss value. log_metrics is to store the metrics to be logged, including loss, top1 and/or top5 accuracies.

Use self.log_dict(log_metrics, on_step, on_epoch, logger) to log the metrics on each step and each epoch. For training, log on each step and each epoch. For validation and testing, only log on each epoch. This way can avoid using on_training_epoch_end() and on_validation_epoch_end().

**validation_step**(*valid_batch*, *batch_idx*) → None

Compute and return the validation loss and metrics on one step.

**test_step**(*test_batch*, *batch_idx*) → None

Compute and return the testing loss and metrics on one step.

**training: bool**

**class** kale.pipeline.base_nn_trainer.**CNNTransformerTrainer**(*feature_extractor*, *task_classifier*, *lr_milestones*, *lr_gamma*, ***kwargs*)

Bases: *BaseNNTrainer*

PyTorch Lightning trainer for cnntransformer.

**Parameters**

- **feature_extractor** (*torch.nn.Sequential, optional*) – the feature extractor network.
- **optimizer** (*dict*) – optimizer parameters.
- **lr_milestones** (*list*) – list of epoch indices. Must be increasing.
- **lr_gamma** (*float*) – multiplicative factor of learning rate decay.

**forward**(*x*)

Forward pass for the model with a feature extractor and a classifier.

**compute_loss**(*batch*, *split_name='valid'*)

Compute loss, top1 and top5 accuracy for a given batch.

**configure_optimizers**()

Set up an SGD optimizer and multistep learning rate scheduler. When self._adapt_lr is True, the learning rate will be decayed by self.lr_gamma every step in milestones.

**training: bool**

**precision: Union[int, str]**

**prepare_data_per_node: bool**

**allow_zero_length_dataloader_with_multiple_devices: bool**

## 12.3 kale.pipeline.deepdta module

**class** kale.pipeline.deepdta.**BaseDTATrainer**(*drug_encoder*, *target_encoder*, *decoder*, *lr=0.001*, *ci_metric=False*, ***kwargs*)

Bases: LightningModule

Base class for all drug target encoder-decoder architecture models, which is based on pytorch lightning wrapper, for more details about pytorch lightning, please check https://github.com/PyTorchLightning/pytorch-lightning. If you inherit from this class, a forward pass function must be implemented.

**Parameters**

- **drug_encoder** – drug information encoder.
- **target_encoder** – target information encoder.
- **decoder** – drug-target representations decoder.
- **lr** – learning rate. (default: 0.001)
- **ci_metric** – calculate the Concordance Index (CI) metric, and the operation is time-consuming for large-scale
- **(default** (*dataset.*) – False)

**configure_optimizers**()

Config adam as default optimizer.

**forward**(*x_drug*, *x_target*)

Same as torch.nn.Module.forward()

**training_step**(*train_batch*, *batch_idx*)

Compute and return the training loss on one step

**validation_step**(*valid_batch*, *batch_idx*)

Compute and return the validation loss on one step

**test_step**(*test_batch*, *batch_idx*)

Compute and return the test loss on one step

**training: bool**

**class** kale.pipeline.deepdta.**DeepDTATrainer**(*drug_encoder*, *target_encoder*, *decoder*, *lr=0.001*, *ci_metric=False*, ***kwargs*)

Bases: *BaseDTATrainer*

An implementation of DeepDTA model based on BaseDTATrainer. :param drug_encoder: drug CNN encoder. :param target_encoder: target CNN encoder. :param decoder: drug-target MLP decoder. :param lr: learning rate.

**forward**(*x_drug*, *x_target*)

Forward propagation in DeepDTA architecture.

**Parameters**

- **x_drug** – drug sequence encoding.
- **x_target** – target protein sequence encoding.

**validation_step**(*valid_batch*, *batch_idx*)

**training: bool**

**precision: Union[int, str]**

**prepare_data_per_node: bool**

**allow_zero_length_dataloader_with_multiple_devices: bool**

## 12.4 kale.pipeline.domain_adapter module

Domain adaptation systems (pipelines) with three types of architectures

This module takes individual modules as input and organises them into an architecture. This is taken directly from https://github.com/criteo-research/pytorch-ada/blob/master/adalib/ada/models/architectures.py with minor changes.

This module uses PyTorch Lightning to standardize the flow.

**class** kale.pipeline.domain_adapter.**GradReverse**(**args*, ***kwargs*)

Bases: Function

The gradient reversal layer (GRL)

This is defined in the DANN paper http://jmlr.org/papers/volume17/15-239/15-239.pdf

Forward pass: identity transformation. Backward propagation: flip the sign of the gradient.

From https://github.com/criteo-research/pytorch-ada/blob/master/adalib/ada/models/layers.py

**static forward**(*ctx*, *x*, *alpha*)

**static backward**(*ctx*, *grad_output*)

kale.pipeline.domain_adapter.**set_requires_grad**(*model*, *requires_grad=True*)

Configure whether gradients are required for a model

kale.pipeline.domain_adapter.**get_aggregated_metrics**(*metric_name_list*, *metric_outputs*)

Get a dictionary of the mean metric values (to log) from metric names and their values

kale.pipeline.domain_adapter.**get_aggregated_metrics_from_dict**(*input_metric_dict*)

Get a dictionary of the mean metric values (to log) from a dictionary of metric values

kale.pipeline.domain_adapter.**get_metrics_from_parameter_dict**(*parameter_dict*, *device*)

Get a key-value pair from the hyperparameter dictionary

**class** kale.pipeline.domain_adapter.**Method**(*value*)

Bases: Enum

Lists the available methods. Provides a few methods that group the methods by type.

**Source = 'Source'**

**DANN = 'DANN'**

**CDAN = 'CDAN'**

**CDAN_E = 'CDAN-E'**

**FSDANN = 'FSDANN'**

**MME = 'MME'**

**WDGRL = 'WDGRL'**

**WDGRLMod = 'WDGRLMod'**

**DAN = 'DAN'**

**JAN = 'JAN'**

**is_mmd_method**()

**is_dann_method**()

**is_cdan_method**()

**is_fewshot_method**()

**allow_supervised**()

kale.pipeline.domain_adapter.**create_mmd_based**(*method:* Method, *dataset*, *feature_extractor*, *task_classifier*, ***train_params*)

MMD-based deep learning methods for domain adaptation: DAN and JAN

kale.pipeline.domain_adapter.**create_dann_like**(*method:* Method, *dataset*, *feature_extractor*, *task_classifier*, *critic*, ***train_params*)

DANN-based deep learning methods for domain adaptation: DANN, CDAN, CDAN+E

kale.pipeline.domain_adapter.**create_fewshot_trainer**(*method:* Method, *dataset*, *feature_extractor*, *task_classifier*, *critic*, ***train_params*)

DANN-based few-shot deep learning methods for domain adaptation: FSDANN, MME

**class** kale.pipeline.domain_adapter.**BaseAdaptTrainer**(*dataset*, *feature_extractor*, *task_classifier*, *method: Optional[str] = None*, *lambda_init: float = 1.0*, *adapt_lambda: bool = True*, *adapt_lr: bool = True*, *nb_init_epochs: int = 10*, *nb_adapt_epochs: int = 50*, *batch_size: int = 32*, *init_lr: float = 0.001*, *optimizer: Optional[dict] = None*)

Bases: LightningModule

Base class for all domain adaptation architectures.

This class implements the classic building blocks used in all the derived architectures for domain adaptation. If you inherit from this class, you will have to implement only:

- a forward pass
- a *compute_loss* function that returns the task loss $\mathcal{L}_c$ and adaptation loss

$\mathcal{L}_a$, as well as a dictionary for summary statistics and other metrics you may want to have access to.

The default training step uses only the task loss $\mathcal{L}_c$ during warmup, then uses the loss defined as:

$\mathcal{L} = \mathcal{L}_c + \lambda \mathcal{L}_a$,

where $\lambda$ will follow the schedule defined by the DANN paper:

$\lambda_p = \frac{2}{1+\exp(-\gamma \cdot p)} - 1$ where $p$ the learning progress changes linearly from 0 to 1.

**Parameters**

- **dataset** (*kale.loaddata.multi_domain*) – the multi-domain datasets to be used for train, validation, and tests.
- **feature_extractor** (*torch.nn.Module*) – the feature extractor network (mapping inputs $x \in \mathcal{X}$ to a latent space $\mathcal{Z}$,).
- **task_classifier** (*torch.nn.Module*) – the task classifier network that learns to predict labels $y \in \mathcal{Y}$ from latent vectors.
- **method** (Method, *optional*) – the method implemented by the class. Defaults to None. Mostly useful when several methods may be implemented using the same class.
- **lambda_init** (*float, optional*) – weight attributed to the adaptation part of the loss. Defaults to 1.0.
- **adapt_lambda** (*bool, optional*) – whether to make lambda grow from 0 to 1 following the schedule from the DANN paper. Defaults to True.
- **adapt_lr** (*bool, optional*) – whether to use the schedule for the learning rate as defined in the DANN paper. Defaults to True.
- **nb_init_epochs** (*int, optional*) – number of warmup epochs (during which lambda=0, training only on the source). Defaults to 10.
- **nb_adapt_epochs** (*int, optional*) – number of training epochs. Defaults to 50.
- **batch_size** (*int, optional*) – defaults to 32.
- **init_lr** (*float, optional*) – initial learning rate. Defaults to 1e-3.
- **optimizer** (*dict, optional*) – optimizer parameters, a dictionary with 2 keys: "type": a string in ("SGD", "Adam", "AdamW") "optim_params": kwargs for the above PyTorch optimizer. Defaults to None.

**property method**

**get_parameters_watch_list**()

Update this list for parameters to watch while training (ie log with MLFlow)

**forward**(*x*)

**compute_loss**(*batch*, *split_name='valid'*)

Define the loss of the model

**Parameters**

- **batch** (*tuple*) – batches returned by the MultiDomainLoader.
- **split_name** (*str, optional*) – learning stage (one of ["train", "valid", "test"]). Defaults to "valid" for validation. "train" is for training and "test" for testing. This is currently used only for naming the metrics used for logging.

**Returns**

a 3-element tuple with task_loss, adv_loss, log_metrics. log_metrics should be a dictionary.

**Raises**
**NotImplementedError** – children of this classes should implement this method.

**training_step**(*batch*, *batch_nb*)

The most generic of training steps

**Parameters**

- **batch** (*tuple*) – the batch as returned by the MultiDomainLoader dataloader iterator: 2 tuples: (x_source, y_source), (x_target, y_target) in the unsupervised setting 3 tuples: (x_source, y_source), (x_target_labeled, y_target_labeled), (x_target_unlabeled, y_target_unlabeled) in the semi-supervised setting
- **batch_nb** (*int*) – id of the current batch.

**Returns**

**must contain a "loss" key with the loss to be used for back-propagation.**
see pytorch-lightning for more details.

**Return type**
dict

**validation_step**(*batch*, *batch_nb*)

**validation_epoch_end**(*outputs*)

**test_step**(*batch*, *batch_nb*)

**test_epoch_end**(*outputs*)

**configure_optimizers**()

**train_dataloader**()

**val_dataloader**()

**test_dataloader**()

**training: bool**

class kale.pipeline.domain_adapter.**BaseDANNLike**(*dataset*, *feature_extractor*, *task_classifier*, *critic*, *alpha=1.0*, *entropy_reg=0.0*, *adapt_reg=True*, *batch_reweighting=False*, ***base_params*)

Bases: *BaseAdaptTrainer*

Common API for DANN-based methods: DANN, CDAN, CDAN+E, WDGRL, MME, FSDANN

**get_parameters_watch_list**()

Update this list for parameters to watch while training (ie log with MLFlow)

**compute_loss**(*batch*, *split_name='valid'*)

**validation_epoch_end**(*outputs*)

**test_epoch_end**(*outputs*)

**training: bool**

**precision: Union[int, str]**

**prepare_data_per_node: bool**

**allow_zero_length_dataloader_with_multiple_devices: bool**

class kale.pipeline.domain_adapter.**DANNTrainer**(*dataset*, *feature_extractor*, *task_classifier*, *critic*, *method=None*, ***base_params*)

Bases: *BaseDANNLike*

This class implements the DANN architecture from Ganin, Yaroslav, et al. "Domain-adversarial training of neural networks." The Journal of Machine Learning Research (2016) https://arxiv.org/abs/1505.07818

**forward**(*x*)

**training: bool**

**precision: Union[int, str]**

**prepare_data_per_node: bool**

**allow_zero_length_dataloader_with_multiple_devices: bool**

class kale.pipeline.domain_adapter.**CDANTrainer**(*dataset*, *feature_extractor*, *task_classifier*, *critic*, *use_entropy=False*, *use_random=False*, *random_dim=1024*, ***base_params*)

Bases: *BaseDANNLike*

Implements CDAN: Long, Mingsheng, et al. "Conditional adversarial domain adaptation." Advances in Neural Information Processing Systems. 2018. https://papers.nips.cc/paper/7436-conditional-adversarial-domain-adaptation.pdf

**forward**(*x*)

**compute_loss**(*batch*, *split_name='valid'*)

**training: bool**

**precision: Union[int, str]**

**prepare_data_per_node: bool**

**allow_zero_length_dataloader_with_multiple_devices: bool**

class kale.pipeline.domain_adapter.**WDGRLTrainer**(*dataset*, *feature_extractor*, *task_classifier*, *critic*, *k_critic=5*, *gamma=10*, *beta_ratio=0*, ***base_params*)

Bases: *BaseDANNLike*

Implements WDGRL as described in Shen, Jian, et al. "Wasserstein distance guided representation learning for domain adaptation." Thirty-Second AAAI Conference on Artificial Intelligence. 2018. https://arxiv.org/pdf/1707.01217.pdf

This class also implements the asymmetric ($eta$) variant described in: Wu, Yifan, et al. "Domain adaptation with asymmetrically-relaxed distribution alignment." ICML (2019) https://arxiv.org/pdf/1903.01689.pdf

**forward**(*x*)

**compute_loss**(*batch*, *split_name='valid'*)

**critic_update_steps**(*batch*)

**training_step**(*batch*, *batch_id*)

**configure_optimizers**()

**training: bool**

**precision: Union[int, str]**

**prepare_data_per_node: bool**

**allow_zero_length_dataloader_with_multiple_devices: bool**

class kale.pipeline.domain_adapter.**WDGRLTrainerMod**(*dataset*, *feature_extractor*, *task_classifier*, *critic*, *k_critic=5*, *gamma=10*, *beta_ratio=0*, ***base_params*)

Bases: *WDGRLTrainer*

Implements a modified version WDGRL as described in Shen, Jian, et al. "Wasserstein distance guided representation learning for domain adaptation." Thirty-Second AAAI Conference on Artificial Intelligence. 2018. https://arxiv.org/pdf/1707.01217.pdf

This class also implements the asymmetric ($eta$) variant described in: Wu, Yifan, et al. "Domain adaptation with asymmetrically-relaxed distribution alignment." ICML (2019) https://arxiv.org/pdf/1903.01689.pdf

**critic_update_steps**(*batch*)

**training_step**(*batch*, *batch_id*, *optimizer_idx*)

**optimizer_step**(*current_epoch*, *batch_nb*, *optimizer*, *optimizer_i*, *second_order_closure=None*, *on_tpu=False*, *using_native_amp=False*, *using_lbfgs=False*)

**configure_optimizers**()

**training: bool**

**precision: Union[int, str]**

**prepare_data_per_node: bool**

**allow_zero_length_dataloader_with_multiple_devices: bool**

class kale.pipeline.domain_adapter.**FewShotDANNTrainer**(*dataset*, *feature_extractor*, *task_classifier*, *critic*, *method*, ***base_params*)

Bases: *BaseDANNLike*

Implements adaptations of DANN to the semi-supervised setting

naive: task classifier is trained on labeled target data, in addition to source data. MME: immplements Saito, Kuniaki, et al. "Semi-supervised domain adaptation via minimax entropy." Proceedings of the IEEE International Conference on Computer Vision. 2019 https://arxiv.org/pdf/1904.06487.pdf

**forward**(*x*)

**compute_loss**(*batch*, *split_name='valid'*)

**training: bool**

**precision: Union[int, str]**

**prepare_data_per_node: bool**

**allow_zero_length_dataloader_with_multiple_devices: bool**

**class** kale.pipeline.domain_adapter.**BaseMMDLike**(*dataset*, *feature_extractor*, *task_classifier*, *kernel_mul=2.0*, *kernel_num=5*, ***base_params*)

Bases: *BaseAdaptTrainer*

Common API for MME-based deep learning DA methods: DAN, JAN

**forward**(*x*)

**compute_loss**(*batch*, *split_name='valid'*)

**validation_epoch_end**(*outputs*)

**test_epoch_end**(*outputs*)

**training: bool**

**precision: Union[int, str]**

**prepare_data_per_node: bool**

**allow_zero_length_dataloader_with_multiple_devices: bool**

**class** kale.pipeline.domain_adapter.**DANTrainer**(*dataset*, *feature_extractor*, *task_classifier*, ***base_params*)

Bases: *BaseMMDLike*

This is an implementation of DAN Long, Mingsheng, et al. "Learning Transferable Features with Deep Adaptation Networks." International Conference on Machine Learning. 2015. http://proceedings.mlr.press/v37/long15.pdf code based on https://github.com/thuml/Xlearn.

**training: bool**

**precision: Union[int, str]**

**prepare_data_per_node: bool**

**allow_zero_length_dataloader_with_multiple_devices: bool**

**class** kale.pipeline.domain_adapter.**JANTrainer**(*dataset*, *feature_extractor*, *task_classifier*, *kernel_mul=(2.0, 2.0)*, *kernel_num=(5, 1)*, ***base_params*)

Bases: *BaseMMDLike*

This is an implementation of JAN Long, Mingsheng, et al. "Deep transfer learning with joint adaptation networks." International Conference on Machine Learning, 2017. https://arxiv.org/pdf/1605.06636.pdf code based on https://github.com/thuml/Xlearn.

**training: bool**

**precision: Union[int, str]**

**prepare_data_per_node: bool**

**allow_zero_length_dataloader_with_multiple_devices: bool**

## 12.5 kale.pipeline.mpca_trainer module

Implementation of MPCA->Feature Selection->Linear SVM/LogisticRegression Pipeline

### References

[1] Swift, A. J., Lu, H., Uthoff, J., Garg, P., Cogliano, M., Taylor, J., … & Kiely, D. G. (2020). A machine learning cardiac magnetic resonance approach to extract disease features and automate pulmonary arterial hypertension diagnosis. European Heart Journal-Cardiovascular Imaging. [2] Song, X., Meng, L., Shi, Q., & Lu, H. (2015, October). Learning tensor-based features for whole-brain fMRI classification. In International Conference on Medical Image Computing and Computer-Assisted Intervention (pp. 613-620). Springer, Cham. [3] Lu, H., Plataniotis, K. N., & Venetsanopoulos, A. N. (2008). MPCA: Multilinear principal component analysis of tensor objects. IEEE Transactions on Neural Networks, 19(1), 18-39.

**class** kale.pipeline.mpca_trainer.**MPCATrainer**(*classifier='svc'*, *classifier_params='auto'*, *classifier_param_grid=None*, *mpca_params=None*, *n_features=None*, *search_params=None*)

Bases: `BaseEstimator`, `ClassifierMixin`

Trainer of pipeline: MPCA->Feature selection->Classifier

**Parameters**

- **classifier** (*str, optional*) – Available classifier options: {“svc”, “linear_svc”, “lr”}, where “svc” trains a support vector classifier, supports both linear and non-linear kernels, optimizes with library “libsvm”; “linear_svc” trains a support vector classifier with linear kernel only, and optimizes with library “liblinear”, which suppose to be faster and better in handling large number of samples; and “lr” trains a classifier with logistic regression. Defaults to “svc”.
- **classifier_params** (*dict, optional*) – Parameters of classifier. Defaults to ‘auto’.
- **classifier_param_grid** (*dict, optional*) – Grids for searching the optimal hyper-parameters. Works only when classifier_params == “auto”. Defaults to None by searching from the following hyper-parameter values: 1. svc, {“kernel”: [“linear”], “C”: [0.0001, 0.001, 0.01, 0.1, 1, 10, 100], “max_iter”: [50000]}, 2. linear_svc, {“C”: [0.0001, 0.001, 0.01, 0.1, 1, 10, 100]}, 3. lr, {“C”: [0.0001, 0.001, 0.01, 0.1, 1, 10, 100]}
- **mpca_params** (*dict, optional*) – Parameters of MPCA, e.g., {“var_ratio”: 0.8}. Defaults to None, i.e., using the default parameters (https://pykale.readthedocs.io/en/latest/kale.embed.html#module-kale.embed.mpca).
- **n_features** (*int, optional*) – Number of features for feature selection. Defaults to None, i.e., all features after dimension reduction will be used.
- **search_params** (*dict, optional*) – Parameters of grid search, for more detail please see https://scikit-learn.org/stable/modules/grid_search.html#grid-search . Defaults to None, i.e., using the default params: {“cv”: 5}.

**fit**(*x*, *y*)

Fit a pipeline with the given data x and labels y

**Parameters**

- **x** (*array-like tensor*) – input data, shape (n_samples, I_1, I_2, …, I_N)
- **y** (*array-like*) – data labels, shape (n_samples, )

**Returns**
self

**predict**(*x*)

Predict the labels for the given data x

**Parameters**
**x** (*array-like tensor*) – input data, shape (n_samples, I_1, I_2, …, I_N)

**Returns**
Predicted labels, shape (n_samples, )

**Return type**
array-like

**decision_function**(*x*)

Decision scores of each class for the given data x

**Parameters**
**x** (*array-like tensor*) – input data, shape (n_samples, I_1, I_2, …, I_N)

**Returns**
decision scores, shape (n_samples,) for binary case, else (n_samples, n_class)

**Return type**
array-like

**predict_proba**(*x*)

Probability of each class for the given data x. Not supported by "linear_svc".

**Parameters**
**x** (*array-like tensor*) – input data, shape (n_samples, I_1, I_2, …, I_N)

**Returns**
probabilities, shape (n_samples, n_class)

**Return type**
array-like

## 12.6 kale.pipeline.multi_domain_adapter module

Multi-source domain adaptation pipelines

kale.pipeline.multi_domain_adapter.**create_ms_adapt_trainer**(*method: str*, *dataset*, *feature_extractor*, *task_classifier*, ***train_params*)

Methods for multi-source domain adaptation

**Parameters**

- **method** (*str*) – Multi-source domain adaptation method, M3SDA or MFSAN
- **dataset** (kale.loaddata.multi_domain.MultiDomainAdapDataset) – the multi-domain datasets to be used for train, validation, and tests.
- **feature_extractor** (*torch.nn.Module*) – feature extractor network
- **task_classifier** (*torch.nn.Module*) – task classifier network

**Returns**
Multi-source domain adaptation trainer.

**Return type**
[pl.LightningModule]

class kale.pipeline.multi_domain_adapter.**BaseMultiSourceTrainer**(*dataset*, *feature_extractor*, *task_classifier*, *n_classes: int*, *target_domain: str*, ***base_params*)

Bases: *BaseAdaptTrainer*

Base class for all domain adaptation architectures

**Parameters**

- **dataset** (*kale.loaddata.multi_domain*) – the multi-domain datasets to be used for train, validation, and tests.
- **feature_extractor** (*torch.nn.Module*) – the feature extractor network
- **task_classifier** (*torch.nn.Module*) – the task classifier network
- **n_classes** (*int*) – number of classes
- **target_domain** (*str*) – target domain name

**forward**(*x*)

**compute_loss**(*batch*, *split_name='valid'*)

**validation_epoch_end**(*outputs*)

**test_epoch_end**(*outputs*)

**training: bool**

**precision: Union[int, str]**

**prepare_data_per_node: bool**

**allow_zero_length_dataloader_with_multiple_devices: bool**

class kale.pipeline.multi_domain_adapter.**M3SDATrainer**(*dataset*, *feature_extractor*, *task_classifier*, *n_classes: int*, *target_domain: str*, *k_moment: int = 3*, ***base_params*)

Bases: *BaseMultiSourceTrainer*

Moment matching for multi-source domain adaptation (M3SDA).

**Reference:**
Peng, X., Bai, Q., Xia, X., Huang, Z., Saenko, K., & Wang, B. (2019). Moment matching for multi-source domain adaptation. In Proceedings of the IEEE/CVF International Conference on Computer Vision (pp. 1406-1415). https://openaccess.thecvf.com/content_ICCV_2019/html/Peng_Moment_Matching_for_Multi-Source_Domain_Adaptation_ICCV_2019_paper.html

**compute_loss**(*batch*, *split_name='valid'*)

**training: bool**

**precision: Union[int, str]**

**prepare_data_per_node: bool**

**allow_zero_length_dataloader_with_multiple_devices: bool**

class kale.pipeline.multi_domain_adapter.**MFSANTrainer**(*dataset*, *feature_extractor*, *task_classifier*, *n_classes: int*, *target_domain: str*, *domain_feat_dim: int = 100*, *kernel_mul: float = 2.0*, *kernel_num: int = 5*, *input_dimension: int = 2*, ***base_params*)

Bases: *BaseMultiSourceTrainer*

Multiple Feature Spaces Adaptation Network (MFSAN)

**Reference: Zhu, Y., Zhuang, F. and Wang, D., 2019, July. Aligning domain-specific distribution and classifier**
for cross-domain classification from multiple sources. In AAAI. https://ojs.aaai.org/index.php/AAAI/article/view/4551

Original implementation: https://github.com/easezyc/deep-transfer-learning/tree/master/MUDA/MFSAN

**compute_loss**(*batch*, *split_name='valid'*)

**cls_discrepancy**(*x*)

Compute discrepancy between all classifiers' probabilistic outputs

**training: bool**

**precision: Union[int, str]**

**prepare_data_per_node: bool**

**allow_zero_length_dataloader_with_multiple_devices: bool**

class kale.pipeline.multi_domain_adapter.**CoIRLS**(*kernel='linear'*, *kernel_kwargs=None*, *alpha=1.0*, *lambda_=1.0*)

Bases: BaseEstimator, ClassifierMixin

Covariate-Independence Regularized Least Squares (CoIRLS)

**Parameters**

- **kernel** (*str, optional*) – {"linear", "rbf", "poly"}. Kernel to use. Defaults to "linear".
- **kernel_kwargs** (*dict or None, optional*) – Hyperparameter for the kernel. Defaults to None.
- **alpha** (*float, optional*) – Hyperparameter of the l2 (Ridge) penalty. Defaults to 1.0.
- **lambda** (*float, optional*) – Hyperparameter of the covariate dependence. Defaults to 1.0.

**Reference:**

**[1] Zhou, S., 2022. Interpretable Domain-Aware Learning for Neuroimage Classification (Doctoral dissertation,**
University of Sheffield).

**[2] Zhou, S., Li, W., Cox, C.R., & Lu, H. (2020). Side Information Dependence as a Regularizer for Analyzing**
Human Brain Conditions across Cognitive Experiments. AAAI 2020, New York, USA.

**fit**(*x*, *y*, *covariates*)

fit a model with input data, labels and covariates

**Parameters**

- **x** (*np.ndarray or tensor*) – shape (n_samples, n_features)
- **y** (*np.ndarray or tensor*) – shape (n_samples, )
- **covariates** (*np.ndarray or tensor*) – (n_samples, n_covariates)

**predict**(*x*)

Predict labels for data x

**Parameters**
**x** (*np.ndarray or tensor*) – Samples need prediction, shape (n_samples, n_features)

**Returns**
Predicted labels, shape (n_samples, )

**Return type**
y (np.ndarray)

**decision_function**(*x*)

Compute decision scores for data x

**Parameters**
**x** (*np.ndarray or tensor*) – Samples need decision scores, shape (n_samples, n_features)

**Returns**
Decision scores, shape (n_samples, )

**Return type**
scores (np.ndarray)

## 12.7 kale.pipeline.video_domain_adapter module

Domain adaptation systems (pipelines) for video data, e.g., for action recognition. Most are inherited from kale.pipeline.domain_adapter.

kale.pipeline.video_domain_adapter.**create_mmd_based_video**(*method:* Method, *dataset*, *image_modality*, *feature_extractor*, *task_classifier*, ***train_params*)

MMD-based deep learning methods for domain adaptation on video data: DAN and JAN

kale.pipeline.video_domain_adapter.**create_dann_like_video**(*method:* Method, *dataset*, *image_modality*, *feature_extractor*, *task_classifier*, *critic*, ***train_params*)

DANN-based deep learning methods for domain adaptation on video data: DANN, CDAN, CDAN+E

**class** kale.pipeline.video_domain_adapter.**BaseMMDLikeVideo**(*dataset*, *image_modality*, *feature_extractor*, *task_classifier*, *kernel_mul=2.0*, *kernel_num=5*, ***base_params*)

Bases: *BaseMMDLike*

Common API for MME-based domain adaptation on video data: DAN, JAN

**forward**(*x*)

**compute_loss**(*batch*, *split_name='valid'*)

**training: bool**

**precision: Union[int, str]**

**prepare_data_per_node: bool**

**allow_zero_length_dataloader_with_multiple_devices: bool**

class kale.pipeline.video_domain_adapter.**DANTrainerVideo**(*dataset*, *image_modality*, *feature_extractor*, *task_classifier*, ***base_params*)

Bases: *BaseMMDLikeVideo*

This is an implementation of DAN for video data.

**training: bool**

**precision: Union[int, str]**

**prepare_data_per_node: bool**

**allow_zero_length_dataloader_with_multiple_devices: bool**

class kale.pipeline.video_domain_adapter.**JANTrainerVideo**(*dataset*, *image_modality*, *feature_extractor*, *task_classifier*, *kernel_mul=(2.0, 2.0)*, *kernel_num=(5, 1)*, ***base_params*)

Bases: *BaseMMDLikeVideo*

This is an implementation of JAN for video data.

**training: bool**

**precision: Union[int, str]**

**prepare_data_per_node: bool**

**allow_zero_length_dataloader_with_multiple_devices: bool**

class kale.pipeline.video_domain_adapter.**DANNTrainerVideo**(*dataset*, *image_modality*, *feature_extractor*, *task_classifier*, *critic*, *method*, ***base_params*)

Bases: *DANNTrainer*

This is an implementation of DANN for video data.

**forward**(*x*)

**compute_loss**(*batch*, *split_name='valid'*)

**training_step**(*batch*, *batch_nb*)

**training: bool**

**precision: Union[int, str]**

**prepare_data_per_node: bool**

**allow_zero_length_dataloader_with_multiple_devices: bool**

class kale.pipeline.video_domain_adapter.**CDANTrainerVideo**(*dataset*, *image_modality*, *feature_extractor*, *task_classifier*, *critic*, *use_entropy=False*, *use_random=False*, *random_dim=1024*, ***base_params*)

Bases: *CDANTrainer*

This is an implementation of CDAN for video data.

**forward**(*x*)

**compute_loss**(*batch*, *split_name='valid'*)

**training: bool**

**precision: Union[int, str]**

**prepare_data_per_node: bool**

**allow_zero_length_dataloader_with_multiple_devices: bool**

class kale.pipeline.video_domain_adapter.**WDGRLTrainerVideo**(*dataset*, *image_modality*, *feature_extractor*, *task_classifier*, *critic*, *k_critic=5*, *gamma=10*, *beta_ratio=0*, ***base_params*)

Bases: *WDGRLTrainer*

This is an implementation of WDGRL for video data.

**forward**(*x*)

**compute_loss**(*batch*, *split_name='valid'*)

**configure_optimizers**()

**critic_update_steps**(*batch*)

**training: bool**

**precision: Union[int, str]**

**prepare_data_per_node: bool**

**allow_zero_length_dataloader_with_multiple_devices: bool**

## 12.8 Module contents

CHAPTER

# THIRTEEN

# UTILITIES

## 13.1 Submodules

## 13.2 kale.utils.download module

Data downloading and compressed data extraction functions, Based on https://github.com/pytorch/vision/blob/master/torchvision/datasets/utils.py https://github.com/pytorch/pytorch/blob/master/torch/hub.py

kale.utils.download.**download_file_by_url**(*url*, *output_directory*, *output_file_name*, *file_format=None*)

Download file/compressed file by url.

**Parameters**

- **url** (*string*) – URL of the object to download
- **output_directory** (*string, optional*) – Full path where object will be saved Abosolute path recommended. Relative path also works.
- **output_file_name** (*string, optional*) – File name which object will be saved as
- **file_format** (*string, optional*) – File format For compressed file, support ["tar.xz", "tar", "tar.gz", "tgz", "gz", "zip"]

**Example: (Grab the raw link from GitHub. Notice that using "raw" in the URL.)**

```
>>> url = "https://github.com/pykale/data/raw/main/videos/video_test_data/ADL/
↪annotations/labels_train_test/adl_P_04_train.pkl"
>>> download_file_by_url(url, "data", "a.pkl", "pkl")
```

```
>>> url = "https://github.com/pykale/data/raw/main/videos/video_test_data.zip"
>>> download_file_by_url(url, "data", "video_test_data.zip", "zip")
```

kale.utils.download.**download_file_gdrive**(*id*, *output_directory*, *output_file_name*, *file_format=None*)

Download file/compressed file by Google Drive id.

**Parameters**

- **id** (*string*) – Google Drive file id of the object to download
- **output_directory** (*string, optional*) – Full path where object will be saved Abosolute path recommended. Relative path also works.
- **output_file_name** (*string, optional*) – File name which object will be saved as

- **file_format** (*string, optional*) – File format For compressed file, support ["tar.xz", "tar", "tar.gz", "tgz", "gz", "zip"]

#### Example

```
>>> gdrive_id = "1U4D23R8u8MJX9KVKb92bZZX-tbpKWtga"
>>> download_file_gdrive(gdrive_id, "data", "demo_datasets.zip", "zip")
```

```
>>> gdrive_id = "1SV7fmAnWj-6AU9X5BGOrvGMoh2Gu9Nih"
>>> download_file_gdrive(gdrive_id, "data", "dummy_data.csv", "csv")
```

## 13.3 kale.utils.logger module

Logging functions, based on https://github.com/HaozhiQi/ISONet/blob/master/isonet/utils/logger.py

kale.utils.logger.**out_file_core**()

Creates an output file name concatenating a formatted date and uuid, but without an extension.

**Returns**
A string to be used in a file name.

**Return type**
string

kale.utils.logger.**construct_logger**(*name*, *save_dir*, *log_to_terminal=False*)

Constructs a logger that saves the output as a text file and optionally logs to the terminal.

The logger is configured to output messages at the DEBUG level, and it saves the output as a text file with a name based on the current timestamp and the specified name. It also saves the output of *git diff HEAD* to a file with the same name and the extension *.gitdiff.patch*.

**Parameters**

- **name** (*str*) – The name of the logger, typically the name of the method being logged.
- **save_dir** (*str*) – The directory where the log file and git diff file will be saved.
- **log_to_terminal** (*bool, optional*) – Whether to also log messages to the terminal. Defaults to False.

**Returns**
The constructed logger.

**Return type**
logging.Logger

**Reference:**
https://docs.python.org/3/library/logging.html

**Raises**
**None.** –

## 13.4 kale.utils.print module

Screen printing functions, from https://github.com/HaozhiQi/ISONet/blob/master/isonet/utils/misc.py

kale.utils.print.**tprint**(**args*)

Temporarily prints things on the screen so that it won't be flooded

kale.utils.print.**pprint**(**args*)

Permanently prints things on the screen to have all info displayed

kale.utils.print.**pprint_without_newline**(**args*)

Permanently prints things on the screen, separated by space rather than newline

## 13.5 kale.utils.seed module

Setting seed for reproducibility

kale.utils.seed.**set_seed**(*seed=1000*)

Sets the seed for generating random numbers to get (as) reproducible (as possible) results.

The CuDNN options are set according to the official PyTorch guidance on reproducibility: https://pytorch.org/docs/stable/notes/randomness.html. Another references are https://discuss.pytorch.org/t/difference-between-torch-manual-seed-and-torch-cuda-manual-seed/13848/6 https://pytorch.org/docs/stable/cuda.html#torch.cuda.manual_seed https://github.com/open-mmlab/mmcv/blob/master/mmcv/runner/utils.py#L58

**Parameters**

**seed** (*int, optional*) – The desired seed. Defaults to 1000.

## 13.6 Module contents

Kale APIs above are ordered following the machine learning pipeline, i.e., functionalities, rather than alphabetically.

CHAPTER

FOURTEEN

# INDICES AND TABLES

- genindex
- modindex
- search

# PYTHON MODULE INDEX

## k

# INDEX

## C

## D



## E



## F

## G

## H



## I



## J



## K

## L



## M

## N



## O



## P

## R



## S

## T

## U



## V

## W