EasyPL

Release 0.3

Alexander Timofeev

Sep 06, 2022

CONTENTS

CHAPTER

ONE

INSTALL GUIDE

You can install this library using pip:

```
pip install easyplib
```

Note: Sorry for the mismatch between the library name in the pypi index and the documentation. The pypi project name normalization algorithms does not allow you to specify an easypl project name.

Also you can install library manually:

```
git clone https://github.com/tam2511/EasyPL.git
cd EasyPL
python setup.py install
```

CHAPTER

TWO

API

2.1 Callbacks

2.1.1 SequentialFinetuning

class easypl.callbacks.finetuners.sequential_tuner.**SequentialFinetuning**(*sequence: Dict*)

Callback for sequence unfreezing model

sequence

Dict of dicts with unfreezing information for epochs. Dict must be like: *{ "epoch_num": EPOCH_INFO, ... }*

EPOCH_INFO: Dict

layers: List
List of layers names, which will be able to unfreeze

lr_gamma: float
Multiple gamma for previous param group learning rate

Type
Dict

Examples

```
>>> from easypl.callbacks import SequentialFinetuning
... sequence = {
...     '0': {
...        'layers': ['block1.layer_name1', ...]
...        },
...     ...
...     '12': {
...         'layers': ['block12.layer_name13', ...]
...         },
...     '14': {
...         'layers': ['block14.layer_name3', ...],
...         'lr_gamma': 0.1
...         }
... }
... finetuner = SequentialFinetuning(sequence=sequence)
```

2.1.2 Loggers

BaseImageLogger

class easypl.callbacks.loggers.base_image.**BaseImageLogger**(*phase: str = 'train', max_samples: int = 1, class_names: Optional[List] = None, mode: str = 'first', sample_key: Optional = None, score_func: Optional[Callable] = None, largest: bool = True, dir_path: Optional[str] = None, save_on_disk: bool = False*)

Abstract callback class for logging images

phase

Phase which will be used by this Logger. Available: [“train”, “val”, “test”, “predict”].

Type
str

max_samples

Maximum number of samples which will be logged at one epoch.

Type
int

class_names

List of class names for pretty logging. If None, then class_names will set range of number of classes.

Type
Optional[List]

mode

Mode of sample generation. Available modes: [“random”, “first”, “top”].

Type
str

sample_key

Key of batch, which define sample. If None, then sample_key will parse *learner.data_keys*.

Type
Optional

score_func

Function for score evaluation. Necessary if “mode” = “top”.

Type
Optional[Callable]

largest

Sorting order for “top” mode

Type
bool

dir_path

If defined, then logs will be writed in this directory. Else in lighting_logs.

Type
Optional[str]

save_on_disk

If true, then logs will be writed on disk to "dir_path".

Type
bool

_post_init(*trainer: Trainer*, *pl_module: LightningModule*)

Method for initialization inverse transforms.

trainer

Trainer of pytorch-lightning

Type
pytorch_lightning.Trainer

pl_module

LightningModule of pytorch-lightning

Type
pytorch_lightning.LightningModule

ClassificationImageLogger

class easypl.callbacks.loggers.image_classification.**ClassificationImageLogger**(*phase: str = 'train'*, *max_samples: int = 1*, *class_names: Optional[List] = None*, *num_classes: Optional[int] = None*, *max_log_classes: Optional[int] = None*, *mode: str = 'first'*, *sample_key: Optional = None*, *score_func: Optional[Callable] = None*, *largest: bool = True*, *dir_path: Optional[str] = None*, *save_on_disk: bool = False*)

Callback class for logging images in classification task

phase

Phase which will be used by this Logger. Available: ["train", "val", "test", "predict"].

Type
str

`max_samples`
Maximum number of samples which will be logged at one epoch.

Type
int

`class_names`
List of class names for pretty logging. If None, then class_names will set range of number of classes.

Type
Optional[List]

`num_classes`
Number of classes. Necessary if *class_names* is None.

Type
Optional[int]

`max_log_classes`
Max of number classes, which will be logged in one sample.

Type
Optional[int]

`mode`
Mode of sample generation. Available modes: [“random”, “first”, “top”].

Type
str

`sample_key`
Key of batch, which define sample. If None, then sample_key will parse *learner.data_keys*.

Type
Optional

`score_func`
Function for score evaluation. Necessary if “mode” = “top”.

Type
Optional[Callable]

`largest`
Sorting order for “top” mode

Type
bool

`dir_path`
If defined, then logs will be writed in this directory. Else in lighting_logs.

Type
Optional[str]

`save_on_disk`
If true, then logs will be writed on disk to “dir_path”.

Type
bool

_log_on_disk(*samples: List*, *dataloader_idx: int = 0*)

Method for logging on disk.

samples

List of returns from *get_log*.

Type
List

dataloader_idx

Index of dataloader.

Type
int, default: 0

_log_tensorboard(*samples: List*, *dataloader_idx: int = 0*)

Method for tensorboard logging.

samples

List of returns from *get_log*.

Type
List

dataloader_idx

Index of dataloader.

Type
int, default: 0

_log_wandb(*samples: List*, *dataloader_idx: int = 0*)

Method for wandb logging.

samples

List of returns from *get_log*.

Type
List

dataloader_idx

Index of dataloader.

Type
int, default: 0

SegmentationImageLogger

class easypl.callbacks.loggers.image_segmentation.**SegmentationImageLogger**(*phase: str = 'train', max_samples: int = 1, class_names: Optional[List] = None, num_classes: Optional[int] = None, max_log_classes: Optional[int] = None, mode: str = 'first', sample_key: Optional = None, score_func: Optional[Callable] = None, largest: bool = True, dir_path: Optional[str] = None, save_on_disk: bool = False, background_class=0, dpi=100*)

Callback class for logging images in segmentation task

phase

Phase which will be used by this Logger. Available: [“train”, “val”, “test”, “predict”].

Type
str

max_samples

Maximum number of samples which will be logged at one epoch.

Type
int

class_names

List of class names for pretty logging. If None, then class_names will set range of number of classes.

Type
Optional[List]

num_classes

Number of classes. Necessary if *class_names* is None.

Type
Optional[int]

max_log_classes

Max of number classes, which will be logged in one sample.

Type
Optional[int]

mode

Mode of sample generation. Available modes: [“random”, “first”, “top”].

Type
str

sample_key

Key of batch, which define sample. If None, then sample_key will parse *learner.data_keys*.

Type
Optional

score_func

Function for score evaluation. Necessary if “mode” = “top”.

Type
Optional[Callable]

largest

Sorting order for “top” mode

Type
bool

dir_path

If defined, then logs will be writed in this directory. Else in lighting_logs.

Type
Optional[str]

save_on_disk

If true, then logs will be writed on disk to “dir_path”.

Type
bool

background_class

Index of background class

Type
int, default: 0

dpi

Dots per inch

Type
int, default: 100

_log_on_disk(*samples: List*, *dataloader_idx: int = 0*)

Method for logging on disk.

samples

List of returns from *get_log*.

Type
List

dataloader_idx

Index of dataloader.

Type
int, default: 0

_log_tensorboard(*samples: List*, *dataloader_idx: int = 0*)

Method for tensorboard logging.

samples

List of returns from *get_log*.

Type
List

dataloader_idx

Index of dataloader.

Type
int, default: 0

_log_wandb(*samples: List*, *dataloader_idx: int = 0*)

Method for wandb logging.

samples

List of returns from *get_log*.

Type
List

dataloader_idx

Index of dataloader.

Type
int, default: 0

DetectionImageLogger

class easypl.callbacks.loggers.image_detection.**DetectionImageLogger**(*phase: str = 'train'*, *max_samples: int = 1*, *class_names: Optional[List] = None*, *num_classes: Optional[int] = None*, *max_detections_per_image: Optional[int] = None*, *confidence: Optional[float] = None*, *mode: str = 'first'*, *sample_key: Optional = None*, *score_func: Optional[Callable] = None*, *largest: bool = True*, *dir_path: Optional[str] = None*, *save_on_disk: bool = False*, *dpi=100*)

Callback class for logging images in detection task

phase

Phase which will be used by this Logger. Available: [“train”, “val”, “test”, “predict”].

Type
str

max_samples

Maximum number of samples which will be logged at one epoch.

Type
int

`class_names`
List of class names for pretty logging. If None, then class_names will set range of number of classes.

Type
Optional[List]

`num_classes`
Number of classes. Necessary if *class_names* is None.

Type
Optional[int]

`max_detections_per_image`
Max of number detections, which will be logged in one sample.

Type
Optional[int]

`confidence`
Min confidence of predictions, which will be logged in one sample.

Type
Optional[float]

`mode`
Mode of sample generation. Available modes: [“random”, “first”, “top”].

Type
str

`sample_key`
Key of batch, which define sample. If None, then sample_key will parse *learner.data_keys*.

Type
Optional

`score_func`
Function for score evaluation. Necessary if “mode” = “top”.

Type
Optional[Callable]

`largest`
Sorting order for “top” mode

Type
bool

`dir_path`
If defined, then logs will be writed in this directory. Else in lighting_logs.

Type
Optional[str]

`save_on_disk`
If true, then logs will be writed on disk to “dir_path”.

Type
bool

_log_on_disk(*samples: List*, *dataloader_idx: int = 0*)

Method for logging on disk.

samples

List of returns from *get_log*.

Type
List

dataloader_idx

Index of dataloader.

Type
int, default: 0

_log_tensorboard(*samples: List*, *dataloader_idx: int = 0*)

Method for tensorboard logging.

samples

List of returns from *get_log*.

Type
List

dataloader_idx

Index of dataloader.

Type
int, default: 0

_log_wandb(*samples: List*, *dataloader_idx: int = 0*)

Method for wandb logging.

samples

List of returns from *get_log*.

Type
List

dataloader_idx

Index of dataloader.

Type
int, default: 0

GANImageLogger

class easypl.callbacks.loggers.image_gan.**GANImageLogger**(*phase: str = 'train'*, *max_samples: int = 1*, *mode: str = 'first'*, *sample_key: Optional = None*, *score_func: Optional[Callable] = None*, *largest: bool = True*, *dir_path: Optional[str] = None*, *save_on_disk: bool = False*, *dpi=100*)

Callback class for logging images in gan task

phase

Phase which will be used by this Logger. Available: [“train”, “val”, “test”, “predict”].

Type
str

max_samples

Maximum number of samples which will be logged at one epoch.

Type
int

mode

Mode of sample generation. Available modes: [“random”, “first”, “top”].

Type
str

sample_key

Key of batch, which define sample. If None, then sample_key will parse *learner.data_keys*.

Type
Optional

score_func

Function for score evaluation. Necessary if “mode” = “top”.

Type
Optional[Callable]

largest

Sorting order for “top” mode

Type
bool

dir_path

If defined, then logs will be writed in this directory. Else in lighting_logs.

Type
Optional[str]

save_on_disk

If true, then logs will be writed on disk to “dir_path”.

Type
bool

_log_on_disk(*samples: List*, *dataloader_idx: int = 0*)

Method for logging on disk.

samples

List of returns from *get_log*.

Type
List

dataloader_idx

Index of dataloader.

Type
int, default: 0

_log_tensorboard(*samples: List*, *dataloader_idx: int = 0*)

Method for tensorboard logging.

samples

List of returns from *get_log*.

Type
List

dataloader_idx

Index of dataloader.

Type
int, default: 0

_log_wandb(*samples: List*, *dataloader_idx: int = 0*)

Method for wandb logging.

samples

List of returns from *get_log*.

Type
List

dataloader_idx

Index of dataloader.

Type
int, default: 0

class easypl.callbacks.loggers.base.**BaseSampleLogger**(*phase: str = 'train'*, *max_samples: int = 1*, *class_names: Optional[List] = None*, *mode: str = 'first'*, *sample_key: Optional = None*, *score_func: Optional[Callable] = None*, *largest: bool = True*, *dir_path: Optional[str] = None*, *save_on_disk: bool = False*)

Abstract callback class for logging any objects

phase

Phase which will be used by this Logger. Available: [“train”, “val”, “test”, “predict”].

Type
str

max_samples

Maximum number of samples which will be logged at one epoch.

Type
int

class_names

List of class names for pretty logging. If None, then class_names will set range of number of classes.

Type
Optional[List]

mode

Mode of sample generation. Available modes: [“random”, “first”, “top”].

Type
str

sample_key

Key of batch, which define sample. If None, then sample_key will parse *learner.data_keys*.

Type
Optional

`score_func`

Function for score evaluation. Necessary if "mode" = "top".

Type
Optional[Callable]

`largest`

Sorting order for "top" mode

Type
bool

`dir_path`

If defined, then logs will be writed in this directory. Else in lighting_logs.

Type
Optional[str]

`save_on_disk`

If true, then logs will be writed on disk to "dir_path".

Type
bool

`_log_on_disk`(*samples: List*, *dataloader_idx: int = 0*)

Abstract method for logging on disk.

`samples`

List of returns from *get_log*.

Type
List

`dataloader_idx`

Index of dataloader.

Type
int, default: 0

`_log_tensorboard`(*samples: List*, *dataloader_idx: int = 0*)

Abstract method for tensorboard logging.

`samples`

List of returns from *get_log*.

Type
List

`dataloader_idx`

Index of dataloader.

Type
int, default: 0

`_log_wandb`(*samples: List*, *dataloader_idx: int = 0*)

Abstract method for wandb logging.

`samples`

List of returns from *get_log*.

Type
List

dataloader_idx
Index of dataloader.
Type
int, default: 0

_post_init(*trainer: Trainer*, *pl_module: LightningModule*)
Abstract method for initialization in first batch handling.

trainer
Trainer of pytorch-lightning
Type
pytorch_lightning.Trainer

pl_module
LightningModule of pytorch-lightning
Type
pytorch_lightning.LightningModule

2.1.3 Mixers

Mixup

class easypl.callbacks.mixers.mixup.**Mixup**(*on_batch: bool = True*, *alpha: float = 0.4*, *num_workers: int = 1*, *p: float = 0.5*, *domen: str = 'classification'*)
Callback for mixuping data operations

on_batch
If True generate samples from batch otherwise from dataset.
Type
bool

p
Mix probability.
Type
float

num_workers
Number of workers for mixing operation.
Type
int

alpha
Parameter of mixup operation.
Type
float

domen
Name of task, in which will be mixed samples. Available: [“classification, segmentation”].
Type
str

mix(*sample1: Dict*, *sample2: Dict*) → Dict

Mixup method for two samples.

sample1

Sample of batch, which will be sampled with sample from *sample2*.

Type
Dict

sample2

Sample from batch or dataset.

Type
Dict

Returns
Mixed sample.

Return type
Dict

Cutmix

class easypl.callbacks.mixers.cutmix.**Cutmix**(*on_batch: bool = True*, *alpha: float = 0.4*, *num_workers: int = 1*, *p: float = 0.5*, *domen: str = 'classification'*)

Callback for cutmixing data operations

on_batch

If True generate samples from batch otherwise from dataset.

Type
bool

p

Mix probability.

Type
float

num_workers

Number of workers for mixing operation.

Type
int

alpha

Parameter of cutmix operation.

Type
float

domen

Name of task, in which will be mixed samples. Available: [“classification, segmentation”].

Type
str

mix(*sample1: Dict*, *sample2: Dict*) → Dict

Cutmix method for two samples.

sample1

Sample of batch, which will be sampled with sample from *sample2*.

Type
Dict

sample2

Sample from batch or dataset.

Type
Dict

Returns
Mixed sample.

Return type
Dict

Mosaic

class easypl.callbacks.mixers.mosaic.**Mosaic**(*on_batch: bool = True*, *n_mosaics: Union[int, List[int]] = 4*, *num_workers: int = 1*, *p: float = 0.5*, *domen: str = 'classification'*)

Callback for mosaic data operations

on_batch

If True generate samples from batch otherwise from dataset.

Type
bool

p

Mix probability.

Type
float

num_workers

Number of workers for mixing operation.

Type
int

n_mosaics

Number of mosaics. If it's list, then number of mosaic will be selected from this list.

Type
Union[int, List[int]]

domen

Name of task, in which will be mixed samples. Available: ["classification, segmentation"].

Type
str

mix(*sample1: Dict*, *sample2: Dict*) → Dict

Mosaic mix method for two samples.

sample1

Sample of batch, which will be sampled with samples from *sample2*.

Type
Dict

sample2

Samples from batch or dataset.

Type
Dict

Returns
Mixed sample.

Return type
Dict

class easypl.callbacks.mixers.base.**MixBaseCallback**(*on_batch=True*, *samples_per: Union[int, List[int]] = 1*, *p: float = 0.5*, *num_workers: int = 1*)

Abstract callback for mixing data operations

on_batch

If True generate samples from batch otherwise from dataset.

Type
bool

samples_per

Number generating samples for one sample.

Type
Union[int, List[int]]

p

Mix probability.

Type
float

num_workers

Number of workers for mixing operation.

Type
int

mix(*sample1: Dict*, *sample2: Dict*) → Dict

Abstract method for mix operation of two samples. *sample2* can be list of samples.

sample1

Sample of batch, which will be sampled with sample/samples from *sample2*.

Type
Dict

sample2

Sample/samples from batch or dataset.

Type
Dict

Returns
Mixed sample.

Return type
Dict

2.1.4 Predictors

BaseImageTestTimeAugmentation

class easypl.callbacks.predictors.base_image.**BaseImageTestTimeAugmentation**(*n: int*, *augmentations: List*, *augmentation_method: str = 'first'*, *phase='val'*)

Abstract image base callback for test-time-augmentation.

n

Number of augmentations.

Type
int

augmentations

List of augmentations, which will be used.

Type
List

augmentation_method

Method of selecting augmentations from list. Available: [“first”, “random”]

Type
str

phase

Phase which will be used by this predictor callback. Available: [“val”, “test”, “predict”].

Type
str

post_init(*trainer: Trainer*, *pl_module: LightningModule*)

Method for initialization in first batch handling. [NOT REQUIRED]

trainer

Trainer of pytorch-lightning

Type
pytorch_lightning.Trainer

pl_module

LightningModule of pytorch-lightning

Type
pytorch_lightning.LightningModule

ClassificationImageTestTimeAugmentation

class easypl.callbacks.predictors.image_classification.**ClassificationImageTestTimeAugmentation**(*n: int*, *augmentations: List*, *augmentation_metho str = 'first'*, *phase='val re- duce_meth Union[str, Callable] = 'mean'*)

Image classification callback for test-time-augmentation

n

Number of augmentations.

Type
int

augmentations

List of augmentations, which will be used.

Type
List

augmentation_method

Method of selecting augmentations from list. Available: [“first”, “random”]

Type
str

phase

Phase which will be used by this predictor callback. Available: [“val”, “test”, “predict”].

Type
str

reduce_method

Method of result reducing

Type
Union[str, Callable]

augment(*sample: Dict*, *augmentation*) → Dict

Method for augmentation apply.

sample

Any sample of batch

Type
Dict

augmentation

Transform object

Returns
Augmented sample

Return type
Dict

metric_formatting(*outputs: Any*, *targets: Any*) → Tuple

Preparing before metric pass.

outputs

Output from model

Type
Any

targets

Targets from batch

Type
Any

Returns
Formatted outputs and targets

Return type
Tuple

post_init(*trainer: Trainer*, *pl_module: LightningModule*)

Method for initialization in first batch handling. [NOT REQUIRED]

trainer

Trainer of pytorch-lightning

Type
pytorch_lightning.Trainer

pl_module

LightningModule of pytorch-lightning

Type
pytorch_lightning.LightningModule

postprocessing(*sample: Dict*, *dataloader_idx: int = 0*) → Dict

Method for postprocessing sample

sample

Any sample of batch

Type
Dict

dataloader_idx

Index of dataloader

Type
int

Returns
Postprocessed sample

Return type
Dict

preprocessing(*sample: Dict*, *dataloader_idx: int = 0*) → Dict

Method for preprocessing sample

sample

Any sample of batch

Type
Dict

dataloader_idx

Index of dataloader

Type
int

Returns
Preprocessed sample

Return type
Dict

reduce(*tensor: Tensor*) → Tensor

Method for reducing of results.

tensor

Any tensor with size [batch_size X . . .]

Type
torch.Tensor

Returns
Reduced tensor

Return type
torch.Tensor

class easypl.callbacks.predictors.base.**BaseTestTimeAugmentation**(*n: int*, *augmentations: List*, *augmentation_method: str = 'first'*, *phase='val'*)

Base callback for test-time-augmentation

n

Number of augmentations.

Type
int

`augmentations`

List of augmentations, which will be used.

Type
List

`augmentation_method`

Method of selecting augmentations from list. Available: ["first", "random"]

Type
str

`phase`

Phase which will be used by this predictor callback. Available: ["val", "test", "predict"].

Type
str

`augment`(*sample: Dict*, *augmentation*) → Dict

Abstract method for augmentation apply.

`sample`

Any sample of batch
Type
Dict

`augmentation`

Transform object

Returns
Augmented sample

Return type
Dict

`metric_formatting`(*outputs: Any*, *targets: Any*) → Tuple

Preparing before metric pass. On default, return passed values.

`outputs`

Output from model
Type
Any

`targets`

Targets from batch
Type
Any

Returns
Formatted outputs and targets

Return type
Tuple

`post_init`(*trainer: Trainer*, *pl_module: LightningModule*)

Abstract method for initialization in first batch handling. [NOT REQUIRED]

trainer

Trainer of pytorch-lightning

Type
pytorch_lightning.Trainer

pl_module

LightningModule of pytorch-lightning

Type
pytorch_lightning.LightningModule

postprocessing(*sample: Dict*, *dataloader_idx: int = 0*) → Dict

Abstract method for postprocessing sample

sample

Any sample of batch

Type
Dict

dataloader_idx

Index of dataloader

Type
int

Returns
Postprocessed sample

Return type
Dict

preprocessing(*sample: Dict*, *dataloader_idx: int = 0*) → Dict

Abstract method for preprocessing sample

sample

Any sample of batch

Type
Dict

dataloader_idx

Index of dataloader

Type
int

Returns
Preprocessed sample

Return type
Dict

reduce(*tensor: Tensor*) → Tensor

Abstract method for reducing of results.

tensor

Any tensor with size [batch_size X . . .]

Type
torch.Tensor

Returns
Reduced tensor

Return type
torch.Tensor

2.2 Datasets

For EasyPL to work correctly, your dataset must return a dict. For ease of creating such a class, we prepared the base class PathBaseDataset.

class easypl.datasets.base.**PathBaseDataset**(*image_prefix: str = ''*, *path_transform: Optional[Callable] = None*, *transform: Optional = None*)

Abstract class of path based dataset

image_prefix

path prefix which will be added to paths of images in csv file

Type
str

path_transform

None or function for transform of path. Will be os.path.join(image_prefix, path_transform(image_path))

Type
Optional[Callable]

transform

albumentations transform class or None

Type
Optional

__len__() → int

Return length of dataset

Return type
int

__getitem__(*idx: int*) → Dict

Read object of dataset by index

idx

index of object in dataset

Type
int

Returns
object of dataset if dict format

Return type
Dict

_read_image(*image_id: Any*, *image_prefix: Optional[str] = None*, ***image_read_kwargs*) → Any

Read image from disk

image_id

Any image identifier

Type
Any

image_prefix

prefix identifier with os.path.join if is str

Type
Optional

image_read_kwargs

Additional arguments for function *easypl.datasets.utils.read_image*

Returns
image object

Return type
Any

For correctly using *PathBaseDataset* you should override *__len__* and *__getitem__* methods. You can use *_read_image* method for simply image loading.

We create simply examples of datasets for classification and segmentation tasks. See below.

2.2.1 Classification

CSVDatasetClassification

class easypl.datasets.classification.csv.**CSVDatasetClassification**(*csv_path: str*, *image_prefix: str = ''*, *path_transform: Optional[Callable] = None*, *transform=None*, *return_label: bool = True*, *image_column: Optional[str] = None*, *target_columns: Optional[Union[str, List[str]]] = None*)

Csv dataset for classfication

csv_path

path to csv file with paths of images

Type
str

return_label

if True return dict with two keys (image, target), else return dict with one key (image)

Type
bool

image_column

column name or None. If None then will be getting the first column

Type
Optional[str]

target_columns

column name/names or None. If None then will be getting all but the first column

Type
Optional[Union[str, List[str]]]

image_prefix

path prefix which will be added to paths of images in csv file

Type
str

path_transform

None or function for transform of path. Will be os.path.join(image_prefix, path_transform(image_path))

Type
Optional[Callable]

transform

albumentations transform class or None

Type
Optional

__len__() → int

Return length of dataset

Return type
int

__getitem__(*idx: int*) → Dict

Read object of dataset by index

idx

index of object in dataset

Type
int

Returns
{"image": ...} or {"image": ..., "target": ...}

Return type
Dict

DirDatasetClassification

class easypl.datasets.classification.dir.**DirDatasetClassification**(*root_path: str*, *label_parser: Callable*, *transform: Optional = None*, *return_label: bool = True*)

Dataset implementation for images in directory on disk (stored images paths in RAM). Require root_path/.../image_path structure.

root_path

path of directory with images

Type
str

transform

albumentations transform or None

Type
Optional

return_label

if True return dict with two keys (image, target), else return dict with one key (image)

Type
bool

label_parser

function for parsing label from relative path

Type
Callable

__len__() → int

Return length of dataset

Return type
int

__getitem__(*idx: int*) → Dict

Read object of dataset by index

idx

index of object in dataset

Type
int

Returns
{"image": ...} or {"image": ..., "target": ...}

Return type
Dict

2.2.2 Segmentation

CSVDatasetSegmentation

class easypl.datasets.segmentation.csv.**CSVDatasetSegmentation**(*csv_path: str*, *image_prefix: str = ''*, *mask_prefix: str = ''*, *path_transform: Optional[Callable] = None*, *transform: Optional = None*, *return_label: bool = True*, *image_column: Optional[str] = None*, *target_column: Optional[str] = None*)

Csv dataset for segmentation

csv_path

path to csv file with paths of images

Type
str

`return_label`
if True return dict with two keys (image, mask), else return dict with one key (image)

Type
bool

`image_column`
column name or None. If None then will be getting the first column

Type
Optional[str]

`target_column`
column name or None. If None then will be getting all but the second column

Type
Optional[str]

`image_prefix`
path prefix which will be added to paths of images in csv file

Type
str

`mask_prefix`
path prefix which will be added to paths of masks in csv file

Type
str

`path_transform`
None or function for transform of path. Will be os.path.join(image_prefix, path_transform(image_path))

Type
Optional[Callable]

`transform`
albumentations transform class or None

Type
Optional

`__len__`() → int
Return length of dataset

Return type
int

`__getitem__`(*idx: int*) → Dict
Read object of dataset by index

`idx`
index of object in dataset
Type
int

Returns
{"image": ...} or {"image": ..., "mask": ...}

Return type
Dict

2.2.3 Detection

CSVDatasetDetection

class easypl.datasets.detection.csv.**CSVDatasetDetection**(*csv_path: str*, *image_prefix: str = ''*, *path_transform: Optional[Callable] = None*, *transform=None*, *return_label: bool = True*, *image_column: Optional[str] = None*, *target_column: Optional[str] = None*)

Csv dataset for detection

csv_path
path to csv file with paths of images

Type
str

return_label
if True return dict with two keys (image, annotations), else return dict with one key (image)

Type
bool

image_column
column name or None. If None then will be getting the first column

Type
Optional[str]

target_column
column name/names or None. If None then will be getting all but the second column

Type
Optional[str]

image_prefix
path prefix which will be added to paths of images in csv file

Type
str

path_transform
None or function for transform of path. Will be os.path.join(image_prefix, path_transform(image_path))

Type
Optional[Callable]

transform
albumentations transform class or None

Type
Optional

`__len__`() → int

Return length of dataset

Return type
int

`__getitem__`(*idx: int*) → Dict

Read object of dataset by index

`idx`

index of object in dataset

Type
int

Returns
{"image": ...} or {"image": ..., "annotations": ...}

Return type
Dict

2.3 Learners

2.3.1 ClassificationLearner

`class` easypl.learners.classification.`ClassificationLearner`(*model: Optional[Union[Module, List[Module]]] = None*, *loss: Optional[Union[Module, List[Module]]] = None*, *optimizer: Optional[Union[WrapperOptimizer, List[WrapperOptimizer]]] = None*, *lr_scheduler: Optional[Union[WrapperScheduler, List[WrapperScheduler]]] = None*, *train_metrics: Optional[List[Metric]] = None*, *val_metrics: Optional[List[Metric]] = None*, *test_metrics: Optional[List[Metric]] = None*, *data_keys: Optional[List[str]] = None*, *target_keys: Optional[List[str]] = None*, *multilabel: bool = False*)

Classification learner.

`model`

torch.nn.Module model.

Type
Optional[Union[torch.nn.Module, List[torch.nn.Module]]]

`loss`

torch.nn.Module loss function.

Type
Optional[Union[torch.nn.Module, List[torch.nn.Module]]]

optimizer

Optimizer wrapper object.

Type
Optional[Union[*WrapperOptimizer*, List[*WrapperOptimizer*]]]

lr_scheduler

Scheduler object for lr scheduling.

Type
Optional[Union[*WrapperScheduler*, List[*WrapperScheduler*]]]

train_metrics

List of train metrics.

Type
Optional[List[Metric]]

val_metrics

List of validation metrics.

Type
Optional[List[Metric]]

test_metrics

List of test metrics.

Type
Optional[List[Metric]]

data_keys

List of data keys

Type
Optional[List[str]]

target_keys

List of target keys

Type
Optional[List[str]]

multilabel

If classification task is multilabel.

Type
bool

forward(*samples: Tensor*) → Tensor

Standart method for forwarding model. .. attribute:: samples

Image tensor.

type
torch.Tensor

Returns
Output from model.

Return type
torch.Tensor

`get_outputs`(*batch: Dict*, *optimizer_idx: int = 0*) → Dict

Abtract method for selecting and preprocessing outputs from batch

`batch`

Batch in step

Type
Dict

`optimizer_idx`

Index of optimizer

Type
int

Returns
Dict with keys: [“loss”, “metric”, “log”]

Return type
Dict

`get_targets`(*batch: Dict*, *optimizer_idx: int = 0*) → Dict

Method for selecting and preprocessing targets from batch

`batch`

Batch in step

Type
Dict

`optimizer_idx`

Index of optimizer

Type
int

Returns
Dict with keys: [“loss”, “metric”, “log”]

Return type
Dict

`loss_step`(*outputs: Tensor*, *targets: Tensor*, *optimizer_idx: int = 0*) → Dict

Method fow loss evaluating.

`outputs`

Outputs from model

Type
torch.Tensor

`targets`

Targets from batch

Type
torch.Tensor

`optimizer_idx`

Index of optimizer

Type
int

Returns
Dict with keys: ["loss", "log"]

Return type
Dict

2.3.2 RecognitionLearner

class easypl.learners.recognition.**RecognitionLearner**(*model: Optional[Union[Module, List[Module]]] = None*, *loss: Optional[Union[Module, List[Module]]] = None*, *optimizer: Optional[Union[WrapperOptimizer, List[WrapperOptimizer]]] = None*, *lr_scheduler: Optional[Union[WrapperScheduler, List[WrapperScheduler]]] = None*, *train_metrics: Optional[List[Metric]] = None*, *val_metrics: Optional[List[Metric]] = None*, *test_metrics: Optional[List[Metric]] = None*, *data_keys: Optional[List[str]] = None*, *target_keys: Optional[List[str]] = None*, *multilabel: bool = False*)

Recognition learner.

model

torch.nn.Module model.

Type
Optional[Union[torch.nn.Module, List[torch.nn.Module]]]

loss

torch.nn.Module loss function.

Type
Optional[Union[torch.nn.Module, List[torch.nn.Module]]]

optimizer

Optimizer wrapper object.

Type
Optional[Union[*WrapperOptimizer*, List[*WrapperOptimizer*]]]

lr_scheduler

Scheduler object for lr scheduling.

Type
Optional[Union[*WrapperScheduler*, List[*WrapperScheduler*]]]

train_metrics

List of train metrics.

Type
Optional[List[Metric]]

val_metrics

List of validation metrics.

Type
Optional[List[Metric]]

`test_metrics`
List of test metrics.

Type
Optional[List[Metric]]

`data_keys`
List of data keys

Type
Optional[List[str]]

`target_keys`
List of target keys

Type
Optional[List[str]]

`multilabel`
If recognition task is multilabel.

Type
bool

`forward`(*samples: Tensor*) → Tensor
Standart method for forwarding model. .. attribute:: samples

Image tensor.

type
torch.Tensor

Returns
Output from model.

Return type
torch.Tensor

`get_outputs`(*batch: Dict*, *optimizer_idx: int = 0*) → Dict
Abtract method for selecting and preprocessing outputs from batch

`batch`
Batch in step
Type
Dict

`optimizer_idx`
Index of optimizer
Type
int

Returns
Dict with keys: [“loss”, “metric”, “log”]

Return type
Dict

`get_targets`(*batch: Dict*, *optimizer_idx: int = 0*) → Dict

Method for selecting and preprocessing targets from batch

`batch`

Batch in step

Type
Dict

`optimizer_idx`

Index of optimizer

Type
int

Returns
Dict with keys: [“loss”, “metric”, “log”]

Return type
Dict

`loss_step`(*outputs: Tensor*, *targets: Tensor*, *optimizer_idx: int = 0*) → Dict

Method fow loss evaluating.

`outputs`

Outputs from model

Type
torch.Tensor

`targets`

Targets from batch

Type
torch.Tensor

`optimizer_idx`

Index of optimizer

Type
int

Returns
Dict with keys: [“loss”, “log”]

Return type
Dict

2.3.3 SegmentationLearner

class easypl.learners.segmentation.**SegmentationLearner**(*model: Optional[Union[Module, List[Module]]] = None*, *loss: Optional[Union[Module, List[Module]]] = None*, *optimizer: Optional[Union[WrapperOptimizer, List[WrapperOptimizer]]] = None*, *lr_scheduler: Optional[Union[WrapperScheduler, List[WrapperScheduler]]] = None*, *train_metrics: Optional[List[Metric]] = None*, *val_metrics: Optional[List[Metric]] = None*, *test_metrics: Optional[List[Metric]] = None*, *data_keys: Optional[List[str]] = None*, *target_keys: Optional[List[str]] = None*, *multilabel: bool = False*)

Segmenatation learner.

model

torch.nn.Module model.

Type
Optional[Union[torch.nn.Module, List[torch.nn.Module]]]

loss

torch.nn.Module loss function.

Type
Optional[Union[torch.nn.Module, List[torch.nn.Module]]]

optimizer

Optimizer wrapper object.

Type
Optional[Union[*WrapperOptimizer*, List[*WrapperOptimizer*]]]

lr_scheduler

Scheduler object for lr scheduling.

Type
Optional[Union[*WrapperScheduler*, List[*WrapperScheduler*]]]

train_metrics

List of train metrics.

Type
Optional[List[Metric]]

val_metrics

List of validation metrics.

Type
Optional[List[Metric]]

test_metrics

List of test metrics.

Type
Optional[List[Metric]]

`data_keys`

List of data keys

Type
Optional[List[str]]

`target_keys`

List of target keys

Type
Optional[List[str]]

`multilabel`

If segmentation task is multilabel.

Type
bool

`forward`(*samples: Tensor*) → Tensor

Standart method for forwarding model. .. attribute:: samples

Image tensor.

type
torch.Tensor

Returns
Output from model.

Return type
torch.Tensor

`get_outputs`(*batch: Dict*, *optimizer_idx: int = 0*) → Dict

Abtract method for selecting and preprocessing outputs from batch

`batch`

Batch in step
Type
Dict

`optimizer_idx`

Index of optimizer
Type
int

Returns
Dict with keys: ["loss", "metric", "log"]

Return type
Dict

`get_targets`(*batch: Dict*, *optimizer_idx: int = 0*) → Dict

Method for selecting and preprocessing targets from batch

`batch`

Batch in step
Type
Dict

optimizer_idx

Index of optimizer

Type
int

Returns
Dict with keys: ["loss", "metric", "log"]

Return type
Dict

loss_step(*outputs: Tensor*, *targets: Tensor*, *optimizer_idx: int = 0*) → Dict

Method fow loss evaluating.

outputs

Outputs from model

Type
torch.Tensor

targets

Targets from batch

Type
torch.Tensor

optimizer_idx

Index of optimizer

Type
int

Returns
Dict with keys: ["loss", "log"]

Return type
Dict

2.3.4 DetectionLearner

class easypl.learners.detection.**DetectionLearner**(*model: Optional[Union[Module, List[Module]]] = None*, *loss: Optional[Union[Module, List[Module]]] = None*, *optimizer: Optional[Union[WrapperOptimizer, List[WrapperOptimizer]]] = None*, *lr_scheduler: Optional[Union[WrapperScheduler, List[WrapperScheduler]]] = None*, *train_metrics: Optional[List[Metric]] = None*, *val_metrics: Optional[List[Metric]] = None*, *test_metrics: Optional[List[Metric]] = None*, *data_keys: Optional[List[str]] = None*, *target_keys: Optional[List[str]] = None*, *image_size_key: Optional[str] = None*, *image_scale_key: Optional[str] = None*, *postprocessing: Optional[BasePostprocessing] = None*)

Detection learner.

`model`

torch.nn.Module model.

Type
Optional[Union[torch.nn.Module, List[torch.nn.Module]]]

`loss`

torch.nn.Module loss function.

Type
Optional[Union[torch.nn.Module, List[torch.nn.Module]]]

`optimizer`

Optimizer wrapper object.

Type
Optional[Union[*WrapperOptimizer*, List[*WrapperOptimizer*]]]

`lr_scheduler`

Scheduler object for lr scheduling.

Type
Optional[Union[*WrapperScheduler*, List[*WrapperScheduler*]]]

`train_metrics`

List of train metrics.

Type
Optional[List[Metric]]

`val_metrics`

List of validation metrics.

Type
Optional[List[Metric]]

`test_metrics`

List of test metrics.

Type
Optional[List[Metric]]

`data_keys`

List of data keys

Type
Optional[List[str]]

`target_keys`

List of target keys

Type
Optional[List[str]]

`postprocessing`

If postprocessing is not None then this

Type
Optional

forward(*samples: Tensor*) → Tensor

Standart method for forwarding model. .. attribute:: samples

Image tensor.

type
torch.Tensor

Returns
Output from model.

Return type
torch.Tensor

get_outputs(*batch: Dict*, *optimizer_idx: int = 0*) → Dict

Abtract method for selecting and preprocessing outputs from batch

batch

Batch in step
Type
Dict

optimizer_idx

Index of optimizer
Type
int

Returns
Dict with keys: ["loss", "metric", "log"]

Return type
Dict

get_targets(*batch: Dict*, *optimizer_idx: int = 0*) → Dict

Method for selecting and preprocessing targets from batch

batch

Batch in step
Type
Dict

optimizer_idx

Index of optimizer
Type
int

Returns
Dict with keys: ["loss", "metric", "log"]

Return type
Dict

loss_step(*outputs: Tensor*, *targets: Tensor*, *optimizer_idx: int = 0*) → Dict

Method fow loss evaluating.

outputs

Outputs from model

Type
torch.Tensor

targets

Targets from batch

Type
torch.Tensor

optimizer_idx

Index of optimizer

Type
int

Returns
Dict with keys: ["loss", "log"]

Return type
Dict

2.3.5 GANLearner

class easypl.learners.gan.**GANLearner**(*model: Optional[List[Module]] = None*, *loss: Optional[List[Module]] = None*, *optimizer: Optional[List[WrapperOptimizer]] = None*, *lr_scheduler: Optional[Union[WrapperScheduler, List[WrapperScheduler]]] = None*, *train_metrics: Optional[List[Metric]] = None*, *val_metrics: Optional[List[Metric]] = None*, *test_metrics: Optional[List[Metric]] = None*, *data_keys: Optional[List[str]] = None*, *target_keys: Optional[List[str]] = None*)

Generative adversarial networks learner.

model

Generative adversarial networks.

Type
Optional[List[torch.nn.Module]]

loss

torch.nn.Module losses function.

Type
Optional[List[torch.nn.Module]]

optimizer

Optimizers wrapper object.

Type
Optional[List[*WrapperOptimizer*]]

lr_scheduler

Scheduler object for lr scheduling.

Type
Optional[Union[*WrapperScheduler*, List[*WrapperScheduler*]]]

`train_metrics`

List of train metrics.

Type
Optional[List[Metric]]

`val_metrics`

List of validation metrics.

Type
Optional[List[Metric]]

`test_metrics`

List of test metrics.

Type
Optional[List[Metric]]

`data_keys`

List of data keys

Type
Optional[List[str]]

`target_keys`

List of target keys

Type
Optional[List[str]]

`forward`(*samples: Tensor*) → Tensor

Standart method for forwarding model. .. attribute:: samples

Image tensor.

type
torch.Tensor

Returns
Output from model.

Return type
torch.Tensor

`get_outputs`(*batch: Dict*, *optimizer_idx: int = 0*) → Dict

Abtract method for selecting and preprocessing outputs from batch

`batch`

Batch in step
Type
Dict

`optimizer_idx`

Index of optimizer
Type
int

Returns
Dict with keys: ["loss", "metric", "log"]

Return type
Dict

`get_targets`(*batch: Dict*, *optimizer_idx: int = 0*) → Dict

Method for selecting and preprocessing targets from batch

`batch`

Batch in step
Type
Dict

`optimizer_idx`

Index of optimizer
Type
int

Returns
Dict with keys: ["loss", "metric", "log"]

Return type
Dict

`loss_step`(*outputs: Dict*, *targets: Dict*, *optimizer_idx: int = 0*) → Dict

Method fow loss evaluating.

`outputs`

Outputs from model
Type
torch.Tensor

`targets`

Targets from batch
Type
torch.Tensor

`optimizer_idx`

Index of optimizer
Type
int

Returns
Dict with keys: ["loss", "log"]

Return type
Dict

`class easypl.learners.base.BaseLearner`(*model: Optional[Union[Module, List[Module]]] = None*, *loss: Optional[Union[Module, List[Module]]] = None*, *optimizer: Optional[Union[WrapperOptimizer, List[WrapperOptimizer]]] = None*, *lr_scheduler: Optional[Union[WrapperScheduler, List[WrapperScheduler]]] = None*, *train_metrics: Optional[List[Metric]] = None*, *val_metrics: Optional[List[Metric]] = None*, *test_metrics: Optional[List[Metric]] = None*, *data_keys: Optional[List[str]] = None*, *target_keys: Optional[List[str]] = None*)

Abstract base learner

`model`

torch.nn.Module model.

Type
Optional[Union[torch.nn.Module, List[torch.nn.Module]]]

`loss`

torch.nn.Module loss function.

Type
Optional[Union[torch.nn.Module, List[torch.nn.Module]]]

`optimizer`

Optimizer wrapper object.

Type
Optional[Union[*WrapperOptimizer*, List[*WrapperOptimizer*]]]

`lr_scheduler`

Scheduler object for lr scheduling.

Type
Optional[Union[*WrapperScheduler*, List[*WrapperScheduler*]]]

`train_metrics`

List of train metrics.

Type
Optional[List[Metric]]

`val_metrics`

List of validation metrics.

Type
Optional[List[Metric]]

`test_metrics`

List of test metrics.

Type
Optional[List[Metric]]

`data_keys`

List of data keys

Type
Optional[List[str]]

`target_keys`

List of target keys

Type
Optional[List[str]]

`get_outputs`(*batch: Dict*, *optimizer_idx: int = 0*) → Dict

Abtract method for selecting and preprocessing outputs from batch

`batch`

Batch in step
Type
Dict

Returns
Dict with keys: ["loss", "metric", "log"]

Return type
Dict

`get_targets`(*batch: Dict*, *optimizer_idx: int = 0*) → Dict

Abtract method for selecting and preprocessing targets from batch

`batch`

Batch in step
Type
Dict

Returns
Dict with keys: ["loss", "metric", "log"]

Return type
Dict

`loss_step`(*outputs: Any*, *targets: Any*, *optimizer_idx: int = 0*) → Dict

Abstract method fow loss evaluating.

`outputs`

Any outputs from model
Type
Any

`targets`

Any targets from batch
Type
Any

Returns
Dict with keys: ["loss", "log"]

Return type
Dict

2.4 Losses

2.4.1 Segmentation Losses

DiceLoss

`class easypl.losses.segmentation.diceloss.DiceLoss`(*weight: Optional[Tensor] = None*, *ignore_index: Optional[int] = None*, ***kwargs*)

Dice loss, need one hot encode input. Taken from: *https://github.com/hubutui/DiceLoss-PyTorch/blob/master/loss.py*. Inputs must be a tensor of shape [N, C, *].

`weight`

An array of shape [num_classes,].

Type
Optional[torch.Tensor]

ignore_index
Class index to ignore.

Type
Optional[int]

kwargs
Additional arguments.

Returns
Loss value

Return type
torch.Tensor

2.5 Lr schedulers

class easypl.lr_schedulers.wrapper.**WrapperScheduler**(*scheduler_class*, ***kwargs*)

Wrapper for pytorch Learning rate Scheduler class.

scheduler_class
Pytorch Learning rate Scheduler class.

__call__(*optimizer: Optimizer*)
Return lr_scheduler

optimizer
WrapperOptimizer object
Type
Optimizer

2.6 Metrics

2.6.1 Classification metrics

SearchAccuracy

class easypl.metrics.classification.search_accuracy.**SearchAccuracy**(*k: Union[int, List] = 1*, *batch_size: int = 512*, *distance: Union[str, Callable] = 'L2'*, *largest: bool = True*, *dist_sync_on_step: bool = False*, *compute_on_step: bool = True*)

Version of accuracy for search case

k

SearchAccuracy return top k (top (k[0], k[1], ...) if k is list) accuracy rate.

Type
Union[int, List]

batch_size

Batch size for evaluate distance operations.

Type
int

distance

Name or function of distance.

Type
Union[str, Callable]

largest

If True metric evaluate top largest samples, else evaluate smallest samples.

Type
bool

SearchMAP

class easypl.metrics.classification.search_mean_average_precision.**SearchMAP**(*k: Union[int, list] = 1*, *batch_size: int = 512*, *distance: Union[str, Callable] = 'L2'*, *largest: bool = True*, *dist_sync_on_step: bool = False*, *compute_on_step: bool = True*)

Version of mean average precision for search case

k

SearchMAP return top k (top (k[0], k[1], ...) if k is list) accuracy rate.

Type
Union[int, List]

batch_size

Batch size for evaluate distance operations.

Type
int

distance

Name or function of distance.

Type
Union[str, Callable]

largest

If True metric evaluate top largest samples, else evaluate smallest samples.

Type
bool

2.6.2 Segmentation metrics

Pixel level metrics

Pixel level Accuracy

class easypl.metrics.segmentation.pixel_level.**PixelLevelAccuracy**(*average: str = 'macro'*, *num_classes: int = 0*, *threshold: float = 0.5*)

Pixel-level accuracy segmentation metric.

average

Method of averaging.

Type
str

num_classes

Number of classes.

Type
int

threshold

Threshold for probabilities of pixels.

Type
float

Pixel level Precision

class easypl.metrics.segmentation.pixel_level.**PixelLevelPrecision**(*average: str = 'macro'*, *num_classes: int = 0*, *threshold: float = 0.5*, *epsilon: float = 1e-08*)

Pixel-level precision segmentation metric.

average

Method of averaging.

Type
str

num_classes

Number of classes.

Type
int

threshold

Threshold for probabilities of pixels.

Type
float

epsilon

Epsilon for correct evalating metric.

Type
float

Pixel level Recall

class easypl.metrics.segmentation.pixel_level.**PixelLevelRecall**(*average: str = 'macro'*, *num_classes: int = 0*, *threshold: float = 0.5*, *epsilon: float = 1e-08*)

Pixel-level recall segmentation metric.

average

Method of averaging.

Type
str

num_classes

Number of classes.

Type
int

threshold

Threshold for probabilities of pixels.

Type
float

epsilon

Epsilon for correct evalating metric.

Type
float

Pixel level FBeta

class easypl.metrics.segmentation.pixel_level.**PixelLevelFBeta**(*average: str = 'macro'*, *num_classes: int = 0*, *threshold: float = 0.5*, *beta: float = 1.0*, *epsilon: float = 1e-08*)

Pixel-level f-beta segmentation metric.

average

Method of averaging.

Type
str

num_classes

Number of classes.

Type
int

threshold

Threshold for probabilities of pixels.

Type
float

beta

Param of metric F-beta

Type
float

epsilon

Epsilon for correct evalating metric.

Type
float

Pixel level F1

class easypl.metrics.segmentation.pixel_level.**PixelLevelF1**(*average: str = 'macro'*, *num_classes: int = 0*, *threshold: float = 0.5*, *epsilon: float = 1e-08*)

Pixel-level f1 segmentation metric.

average

Method of averaging.

Type
str

num_classes

Number of classes.

Type
int

threshold

Threshold for probabilities of pixels.

Type
float

epsilon

Epsilon for correct evalating metric.

Type
float

class easypl.metrics.segmentation.pixel_level.**PixelLevelBase**(*average: str = 'macro'*, *num_classes: int = 0*, *threshold: float = 0.5*)

Abstract class for pixel-level segmentation metrics.

average

Method of averaging.

Type
str

num_classes

Number of classes.

Type
int

threshold

Threshold for probabilities of pixels.

Type
float

2.6.3 Detection metrics

MAP

class easypl.metrics.detection.mean_average_precision.**MAP**(*iou_thresholds: Optional[List[float]] = None*, *rec_thresholds: Optional[List[float]] = None*, *max_detection_thresholds: Optional[List[int]] = None*, *class_metrics: bool = False*, ***kwargs*)

Wrapper for MeanAveragePrecision from torchmetrics.

iou_thresholds

Iou threshold/thresholds for boxes.

Type
Optional[List[float]]

rec_thresholds

Recall thresholds for evaluation.

Type
Optional[List[float]]

max_detection_thresholds

Thresholds on max detections per image.

Type
Optional[List[int]]

class_metrics

Option to enable per-class metrics for mAP and mAR_100.

Type
bool

kwargs

Torchmetrics Metric args.

`reset()`

This method automatically resets the metric state variables to their default value.

BaseDetectionMetric

`class easypl.metrics.detection.base.BaseDetectionMetric(`*iou_threshold: Union[float, List[float]]*, *confidence: Union[float, List[float]]*, *num_classes: Optional[int] = None*, ***kwargs*`)`

Base detection metric. Compute true positive, false negative and false positive metrics.

`iou_threshold`

Iou threshold/thresholds for boxes.

Type
Union[float, List[float]]

`confidence`

Confidence/confidences thresholds.

Type
Union[float, List[float]]

`num_classes`

Number of classes.

Type
Optional[int]

`kwargs`

Torchmetrics Metric args.

`reset()`

This method automatically resets the metric state variables to their default value.

FBetaDetection

`class easypl.metrics.detection.f_beta.FBetaDetection(`*iou_threshold: Union[float, List[float]]*, *confidence: Optional[List[float]] = None*, *num_classes: Optional[int] = None*, *beta: float = 1.0*, *eps: float = 1e-09*, ***kwargs*`)`

Evaluate optimal confidence by F beta metric and return with precision, recall values.

`iou_threshold`

Iou threshold/thresholds for boxes.

Type
Union[float, List[float]]

`confidence`

Confidence/confidences thresholds or None. If is None then evaluate as arange from 0 to 1 with step 0.05.

Type
Optional[List[float]]

num_classes

Number of classes.

Type
Optional[int]

beta

Parameter of F metric.

Type
float

eps

Epsilon.

Type
float

kwargs

Torchmetrics Metric args.

class easypl.metrics.base.**MetricsList**(*dist_sync_on_step: bool = False*, *compute_on_step: bool = True*)

List of metrics

clone()

Make a copy of the metric.

class easypl.metrics.torch.**TorchMetric**(*metric: Metric*, *class_names: Optional[List] = None*)

Wrapper for metrics from torchmetrics

metric

Metric object from torchmetrics.

Type
Metric

class_names

Names of classes.

Type
Optional[List]

Examples

```
>>> from torchmetrics import F1
... from easypl.metrics import TorchMetric
... result_metric = TorchMetric(F1(), class_names=None)
```

2.7 Optimizers

class easypl.optimizers.wrapper.**WrapperOptimizer**(*optimizer_cls*, ***kwargs*)

Wrapper for pytorch Optimizer class.

optimizer_cls

Pytorch Optimizer class.

kwargs

Additional arguments.

__call__(*params*) → Optimizer

Return optimizer.

params

Model params

Returns
Optimizer object

Return type
Optimizer

CHAPTER

THREE

EXAMPLES

3.1 Simple multiclass classification example

We can observe simple example for classification task with two classes (cats and dogs).

First, you should import common libraries and packages below. (If you don't have some package, than install it).

```
import cv2
from torch.utils.data import Dataset, DataLoader
import torch
import torch.optim as optim
import torch.nn as nn
from pytorch_lightning import Trainer
from pytorch_lightning.callbacks import ModelCheckpoint
from albumentations.augmentations import *
from albumentations.core.composition import *
from albumentations.pytorch.transforms import *
from timm import create_model
import random
from torchmetrics import *
import shutil
```

Than you can import EasyPL packages, as like:

```
from easypl.learners import ClassificationLearner
from easypl.metrics import TorchMetric
from easypl.optimizers import WrapperOptimizer
from easypl.lr_schedulers import WrapperScheduler
from easypl.datasets import CSVDatasetClassification
from easypl.callbacks import ClassificationImageLogger
from easypl.callbacks import Cutmix
from easypl.callbacks import ClassificationImageTestTimeAugmentation
```

Than you should define datasets and dataloaders. You can use this simple example:

```
train_transform = Compose([
    HorizontalFlip(p=0.5),
    Rotate(p=0.5),
    LongestMaxSize(max_size=224),
    PadIfNeeded(min_height=224, min_width=224, border_mode=cv2.BORDER_CONSTANT, value=0,␣
↪mask_value=0),
```

(continues on next page)

(continued from previous page)

```
    Normalize(),
    ToTensorV2(),
])

val_transform = Compose([
    LongestMaxSize(max_size=600),
    PadIfNeeded(min_height=600, min_width=600, border_mode=cv2.BORDER_CONSTANT, value=0,␣
↪mask_value=0),
    Normalize(),
    ToTensorV2(),
])

train_dataset = CSVDatasetClassification('../input/cat-dog-test/train.csv', image_prefix=
↪'../input/cat-dog-test/train', transform=train_transform, return_label=True)
val_dataset = CSVDatasetClassification('../input/cat-dog-test/val.csv', image_prefix='../
↪input/cat-dog-test/val', transform=val_transform, return_label=True)

train_dataloader = DataLoader(train_dataset, batch_size=16, shuffle=True, pin_
↪memory=True, num_workers=2)
val_dataloader = DataLoader(train_dataset, batch_size=16, shuffle=False, pin_memory=True,
↪ num_workers=2)
```

Than we should define model (used timm), loss function, optimizer and metrics:

```
model = create_model('resnet18', pretrained=True, num_classes=2)

loss_f = nn.CrossEntropyLoss()

optimizer = WrapperOptimizer(optim.Adam, lr=1e-4)
lr_scheduler = WrapperScheduler(optim.lr_scheduler.StepLR, step_size=2, gamma=1e-1,␣
↪interval='epoch')

train_metrics = []
val_metrics = [TorchMetric(F1(num_classes=2, average='none'), class_names=['cat', 'dog
↪'])]
```

If you need in callbacks, you can use our simple realization. Creating of callbacks looks like:

```
# Logger of outputs (images)
image_logger = ClassificationImageLogger(
    phase='train',
    max_samples=10,
    class_names=['cat', 'dog'],
    max_log_classes=2,
    dir_path='images',
    save_on_disk=True,
)

# Cutmix callback
cutmix = Cutmix(
    on_batch=True,
    p=1.0,
```

(continues on next page)

(continued from previous page)

```
    domen='classification',
)

# Test time augmentation callback
tta = ClassificationImageTestTimeAugmentation(
    n=2,
    augmentations=[VerticalFlip(p=1.0)],
    phase='val'
)
```

In finally, we should define learner and trainer, and than run training.

```
learner = ClassificationLearner(
    model=model,
    loss=loss_f,
    optimizer=optimizer,
    lr_scheduler=lr_scheduler,
    train_metrics=train_metrics,
    val_metrics=val_metrics,
    data_keys=['image'],
    target_keys=['target'],
    multilabel=False
)
trainer = Trainer(
    gpus=1,
    callbacks=[image_logger, cutmix, tta],
    max_epochs=3,
    precision=16
)
trainer.fit(learner, train_dataloaders=train_dataloader, val_dataloaders=[val_
↪dataloader])
```

3.2 Simple semantic segmentation example

We can observe simple example for segmentation task with one class (text).

First, you should import common libraries and packages below. (If you don't have some package, than install it).

```
import cv2
import numpy as np
from torch.utils.data import Dataset, DataLoader
import torch
import torch.optim as optim
import torch.nn as nn
from pytorch_lightning import Trainer
from pytorch_lightning.callbacks import ModelCheckpoint
from albumentations.augmentations import *
from albumentations.core.composition import *
from albumentations.pytorch.transforms import *
from timm import create_model
import random
```

(continues on next page)

(continued from previous page)

```
from torchmetrics import *
import shutil
```

Than you can import EasyPL packages, as like:

```
from easypl.learners import SegmentationLearner
from easypl.metrics import TorchMetric
from easypl.metrics.segmentation import PixelLevelF1
from easypl.optimizers import WrapperOptimizer
from easypl.lr_schedulers import WrapperScheduler
from easypl.datasets import CSVDatasetSegmentation
from easypl.callbacks import SegmentationImageLogger
from easypl.callbacks import Mixup
from easypl.losses.segmentation import DiceLoss
```

Than you should define datasets and dataloaders. You can use this simple example:

```
train_transform = Compose([
    HorizontalFlip(p=0.5),
    Rotate(p=0.5),
    LongestMaxSize(max_size=224),
    PadIfNeeded(min_height=224, min_width=224, border_mode=cv2.BORDER_CONSTANT, value=0,␣
↪mask_value=0),
    Normalize(),
    ToTensorV2(),
])

val_transform = Compose([
    LongestMaxSize(max_size=600),
    PadIfNeeded(min_height=600, min_width=600, border_mode=cv2.BORDER_CONSTANT, value=0,␣
↪mask_value=0),
    Normalize(),
    ToTensorV2(),
])

dataset = CSVDatasetSegmentation(
    csv_path='../input/lan-segmentation-1/train.csv',
    image_prefix='../input/lan-segmentation-1/train/images',
    mask_prefix='../input/lan-segmentation-1/train/masks',
    image_column='path',
    target_column='path'
)

class WrapperDataset(Dataset):
    def __init__(self, dataset, transform=None, idxs=[]):
        self.dataset = dataset
        self.transform = transform
        self.idxs = idxs

    def __getitem__(self, idx):
        idx = self.idxs[idx]
        row = self.dataset[idx]
```

(continues on next page)

(continued from previous page)

```
        row['mask'][row['mask'] < 150] = 0
        row['mask'][row['mask'] > 150] = 1
        if self.transform:
            result = self.transform(image=row['image'], mask=row['mask'])
            row['image'] = result['image']
            row['mask'] = result['mask'].to(dtype=torch.long)
        row['mask'] = one_hot(row['mask'], num_classes=2).permute(2, 0, 1)
        return row

    def __len__(self):
        return len(self.idxs)

image_size = 768

train_transform = A.Compose([
    A.HorizontalFlip(p=0.5),
    A.ColorJitter(p=0.7),
    A.LongestMaxSize(max_size=image_size),
    A.PadIfNeeded(min_height=image_size, min_width=image_size, border_mode=cv2.BORDER_
↪CONSTANT, value=0, mask_value=0),
    A.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]),
    ToTensorV2()
])
val_transform = A.Compose([
    A.LongestMaxSize(max_size=image_size),
    A.PadIfNeeded(min_height=image_size, min_width=image_size, border_mode=cv2.BORDER_
↪CONSTANT, value=0, mask_value=0),
    A.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]),
    ToTensorV2()
])

size_dataset = len(dataset)
val_size = int(size_dataset * 0.1)
train_dataset = WrapperDataset(dataset, transform=train_transform, idxs=list(range(val_
↪size, size_dataset)))
val_dataset = WrapperDataset(dataset, transform=val_transform, idxs=list(range(val_
↪size)))

train_dataloader = DataLoader(train_dataset, batch_size=4, shuffle=True, num_workers=0,␣
↪drop_last=True)
val_dataloader = DataLoader(val_dataset, batch_size=8, shuffle=False, num_workers=0)
```

Than we should define model (used timm), loss function, optimizer and metrics:

```
model = smp.UnetPlusPlus(
    encoder_name="resnet18",
    encoder_weights="imagenet",
    in_channels=3,
    decoder_use_batchnorm=False,
    classes=2,
)
```

(continues on next page)

(continued from previous page)

```
loss_f = DiceLoss(weight=torch.tensor([1, 10]))

optimizer = WrapperOptimizer(optim.Adam, lr=1e-4)

num_epochs = 7
num_gpus = 1

lr_scheduler = WrapperScheduler(
    torch.optim.lr_scheduler.OneCycleLR, max_lr=3e-4, pct_start=1 / (num_epochs),
    total_steps=int(len(train_dataloader) * num_epochs / num_gpus) + 10, div_factor=1e+3,
↪ final_div_factor=1e+4,
    anneal_strategy='cos', interval='step'
)

class_names = ['background', 'text']


train_metrics = [

]

val_metrics = [
    TorchMetric(PixelLevelF1(average='none', num_classes=len(class_names)), class_names),
]
```

If you need in callbacks, you can use our simple realization. Creating of callbacks looks like:

```
# Logger of outputs (images)
logger = SegmentationImageLogger(
    phase='val',
    max_samples=10,
    num_classes=2,
    save_on_disk=True,
    dir_path='images'
)

# Cutmix callback
mixup = Mixup(
    on_batch=True,
    p=1.0,
    domen='segmentation',
)
```

In finally, we should define learner and trainer, and than run training.

```
learner = SegmentationLearner(
    model=model,
    loss=loss_f,
    optimizer=optimizer,
    lr_scheduler=lr_scheduler,
    train_metrics=train_metrics,
    val_metrics=val_metrics,
```

(continues on next page)

(continued from previous page)

```
    data_keys=['image'],
    target_keys=['mask'],
    multilabel=False
)
trainer = pl.Trainer(gpus=num_gpus, callbacks=[logger, mixup, checkpoint_callback], max_
↪epochs=num_epochs)
trainer.fit(learner, train_dataloaders=train_dataloader, val_dataloaders=[val_
↪dataloader])
```

3.3 Simple detection train example

We can observe simple example for detection in pascal dataset using effdet project (https://github.com/rwightman/efficientdet-pytorch).

First, you should import common libraries and packages below. (If you don't have some package, than install it).

```
import cv2
from torch.utils.data import Dataset, DataLoader
import torch
import torch.optim as optim
import torch.nn as nn
from pytorch_lightning import Trainer
from pytorch_lightning.callbacks import ModelCheckpoint
from albumentations.augmentations import *
from albumentations.core.composition import *
from albumentations.pytorch.transforms import *
from timm import create_model
import random
from torchmetrics import *
import shutil
```

Than you can import EasyPL packages, as like:

```
from easypl.learners.detection import DetectionLearner
from easypl.metrics.detection import FBetaDetection
from easypl.optimizers import WrapperOptimizer
from easypl.lr_schedulers import WrapperScheduler
from easypl.datasets import CSVDatasetDetection
from easypl.callbacks.loggers import DetectionImageLogger
from easypl.callbacks.mixers import Mixup, Cutmix, Mosaic
from easypl.callbacks.finetuners import OptimizerInitialization
from easypl.utilities.detection import BasePostprocessing
```

Than you should define datasets and dataloaders. You can use this simple example:

```
train_transform = Compose([
    HorizontalFlip(p=0.5),
    LongestMaxSize(max_size=512),
    PadIfNeeded(min_height=512, min_width=512, border_mode=cv2.BORDER_CONSTANT, value=0,␣
↪mask_value=0),
    Normalize(),
```

(continues on next page)

(continued from previous page)

```
    ToTensorV2(),
], bbox_params=BboxParams(format='pascal_voc', min_visibility=0.1))

val_transform = Compose([
    LongestMaxSize(max_size=512),
    PadIfNeeded(min_height=512, min_width=512, border_mode=cv2.BORDER_CONSTANT, value=0,␣
↪mask_value=0),
    Normalize(),
    ToTensorV2(),
], bbox_params=BboxParams(format='pascal_voc', min_visibility=0.1))

test_transform = Compose([
    Normalize(),
    ToTensorV2(),
], bbox_params=BboxParams(format='pascal_voc', min_visibility=0.1))


def collate_fn(batch):
    images = torch.stack([_['image'] for _ in batch])
    max_anno_size = max(len(_['annotations'][0]) for _ in batch)
    image_sizes = torch.from_numpy(np.stack([_['image_size'] for _ in batch]))
    image_scales = torch.from_numpy(np.stack([_['image_scale'] for _ in batch]))
    annotations = torch.ones(len(batch), max_anno_size, 5, dtype=torch.float) * -1
    for i in range(len(batch)):
        annotations[i][:len(batch[i]['annotations'][0])] = batch[i]['annotations'][0]
    return {
        'image': images,
        'annotations': annotations,
        'image_size': image_sizes,
        'image_scale': image_scales
    }

train_dataset = CSVDatasetDetection('../input/pascal/train.csv', image_prefix='../input/
↪pascal/train', transform=train_transform, return_label=True)
val_dataset = CSVDatasetDetection('../input/pascal/val.csv', image_prefix='../input/
↪pascal/val', transform=val_transform, return_label=True)

train_dataloader = DataLoader(train_dataset, batch_size=16, shuffle=True, pin_
↪memory=True, num_workers=2)
val_dataloader = DataLoader(train_dataset, batch_size=16, shuffle=False, pin_memory=True,
↪ num_workers=2)
```

Than we should define model (used effnet: https://github.com/rwightman/efficientdet-pytorch), loss function, optimizer and metrics:

```
from effdet import EfficientDet, get_efficientdet_config

num_classes = 20

config = get_efficientdet_config('tf_efficientdet_d0')

model = EfficientDet(config, pretrained_backbone=True)
```

(continues on next page)

(continued from previous page)

```
model.reset_head(num_classes=num_classes)

from effdet.anchors import Anchors, AnchorLabeler, generate_detections
from effdet.loss import DetectionLoss

class EfficientDetLoss(nn.Module):
    def __init__(self, model, create_labeler=False):
        super().__init__()
        self.model = model
        self.config = model.config  # FIXME remove this when we can use @property␣
↪(torchscript limitation)
        self.num_levels = model.config.num_levels
        self.num_classes = model.config.num_classes
        self.anchors = Anchors.from_config(model.config)
        self.max_detection_points = model.config.max_detection_points
        self.max_det_per_image = model.config.max_det_per_image
        self.soft_nms = model.config.soft_nms
        self.anchor_labeler = AnchorLabeler(self.anchors, self.num_classes, match_
↪threshold=0.5)
        self.loss_fn = DetectionLoss(model.config)

    def forward(self, x, target):
        class_out, box_out = x
        cls_targets, box_targets, num_positives = self.anchor_labeler.batch_label_
↪anchors(target[:, :, :4], target[:, :, 4])
        loss, class_loss, box_loss = self.loss_fn(class_out, box_out, cls_targets, box_
↪targets, num_positives)
        output = {'loss': loss, 'class_loss': class_loss, 'box_loss': box_loss}
        return output


loss_f = EfficientDetLoss(model=model, create_labeler=True)

num_epochs = 5
num_gpus = 1

optimizer = WrapperOptimizer(optim.Adam, lr=1e-4)


lr_scheduler = WrapperScheduler(
    torch.optim.lr_scheduler.OneCycleLR, max_lr=3e-4, pct_start=1 / (num_epochs),
    total_steps=int(len(train_dataloader) * num_epochs / num_gpus) + 10, div_factor=1e+3,
↪ final_div_factor=1e+4,
    anneal_strategy='cos', interval='step'
)

train_metrics = []
val_metrics = [FBetaDetection([0.5])]
```

If you need in callbacks, you can use our simple realization. Creating of callbacks looks like:

```
# Logger of outputs (images)
```

(continues on next page)

(continued from previous page)

```
image_logger = DetectionImageLogger(phase='val', num_classes=num_classes)
```

In finally, we should define learner and trainer, and than run training.

```
learner = DetectionLearner(
    model=model,
    loss=loss_f,
    optimizer=optimizer,
    lr_scheduler=lr_scheduler,
    train_metrics=train_metrics,
    val_metrics=val_metrics,
    data_keys=['image'],
    target_keys=['annotations'],
    postprocessing=EfficientdetPostprocessing(model),
    image_size_key='image_size',
    image_scale_key='image_scale'
)
trainer = Trainer(
    accelerator='gpu',
    devices=1,
    callbacks=[image_logger],
    max_epochs=num_epochs,
#     precision=32
)
trainer.fit(learner, train_dataloaders=train_dataloader, val_dataloaders=[val_
↪dataloader])
```

Above there are few examples of code, which you can use for yourself projects.

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search

INDEX

Symbols

A

B

C

D

E

F

G

I

K

L

M

N

O

P

R

S

T

V

W