Open Metric Learning

Shabanov Aleksei

Apr 25, 2024

OML

OML is a PyTorch-based framework to train and validate the models producing high-quality embeddings.

CHAPTER

ONE

INSTALLATION

OML is available in PyPI:

```
pip install -U open-metric-learning
```

You can also pull the prepared image from DockerHub…

```
docker pull omlteam/oml:gpu
docker pull omlteam/oml:cpu
```

CHAPTER

TWO

FAQ

You may think *"If I need image embeddings I can simply train a vanilla classifier and take its penultimate layer"*. Well, it makes sense as a starting point. But there are several possible drawbacks:

- If you want to use embeddings to perform searching you need to calculate some distance among them (for example, cosine or L2). Usually, **you don't directly optimize these distances during the training** in the classification setup. So, you can only hope that final embeddings will have the desired properties.
- **The second problem is the validation process**. In the searching setup, you usually care how related your top-N outputs are to the query. The natural way to evaluate the model is to simulate searching requests to the reference set and apply one of the retrieval metrics. So, there is no guarantee that classification accuracy will correlate with these metrics.
- Finally, you may want to implement a metric learning pipeline by yourself. **There is a lot of work**: to use triplet loss you need to form batches in a specific way, implement different kinds of triplets mining, tracking distances, etc. For the validation, you also need to implement retrieval metrics, which include effective embeddings accumulation during the epoch, covering corner cases, etc. It's even harder if you have several gpus and use DDP. You may also want to visualize your search requests by highlighting good and bad search results. Instead of doing it by yourself, you can simply use OML for your purposes.

PML is the popular library for Metric Learning, and it includes a rich collection of losses, miners, distances, and reducers; that is why we provide straightforward examples of using them with OML. Initially, we tried to use PML, but in the end, we came up with our library, which is more pipeline / recipes oriented. That is how OML differs from PML:

- OML has Pipelines which allows training models by preparing a config and your data in the required format (it's like converting data into COCO format to train a detector from mmdetection).
- OML focuses on end-to-end pipelines and practical use cases. It has config based examples on popular benchmarks close to real life (like photos of products of thousands ids). We found some good combinations of hyperparameters on these datasets, trained and published models and their configs. Thus, it makes OML more recipes oriented than PML, and its author confirms this saying that his library is a set of tools rather the recipes, moreover, the examples in PML are mostly for CIFAR and MNIST datasets.
- OML has the Zoo of pretrained models that can be easily accessed from the code in the same way as in `torchvision` (when you type `resnet50(pretrained=True)`).
- OML is integrated with PyTorch Lightning, so, we can use the power of its Trainer. This is especially helpful when we work with DDP, so, you compare our DDP example and the PMLs one. By the way, PML also has Trainers, but it's not widely used in the examples and custom `train` / `test` functions are used instead.

We believe that having Pipelines, laconic examples, and Zoo of pretrained models sets the entry threshold to a really low value.

Metric Learning problem (also known as *extreme classification* problem) means a situation in which we have thousands of ids of some entities, but only a few samples for every entity. Often we assume that during the test stage (or production)

we will deal with unseen entities which makes it impossible to apply the vanilla classification pipeline directly. In many cases obtained embeddings are used to perform search or matching procedures over them.

Here are a few examples of such tasks from the computer vision sphere:

- Person/Animal Re-Identification
- Face Recognition
- Landmark Recognition
- Searching engines for online shops

and many others.

- `embedding` - model's output (also known as `features vector` or `descriptor`).
- `query` - a sample which is used as a request in the retrieval procedure.
- `gallery set` - the set of entities to search items similar to `query` (also known as `reference` or `index`).
- `Sampler` - an argument for `DataLoader` which is used to form batches
- `Miner` - the object to form pairs or triplets after the batch was formed by `Sampler`. It's not necessary to form the combinations of samples only inside the current batch, thus, the memory bank may be a part of `Miner`.
- `Samples/Labels/Instances` - as an example let's consider DeepFashion dataset. It includes thousands of fashion item ids (we name them `labels`) and several photos for each item id (we name the individual photo as `instance` or `sample`). All of the fashion item ids have their groups like "skirts", "jackets", "shorts" and so on (we name them `categories`). Note, we avoid using the term `class` to avoid misunderstanding.
- `training epoch` - batch samplers which we use for combination-based losses usually have a length equal to `[number of labels in training dataset] / [numbers of labels in one batch]`. It means that we don't observe all of the available training samples in one epoch (as opposed to vanilla classification), instead, we observe all of the available labels.

It may be comparable with the current (2022 year) SotA methods, for example, Hyp-ViT. *(Few words about this approach: it's a ViT architecture trained with contrastive loss, but the embeddings were projected into some hyperbolic space. As the authors claimed, such a space is able to describe the nested structure of real-world data. So, the paper requires some heavy math to adapt the usual operations for the hyperbolical space.)*

We trained the same architecture with triplet loss, fixing the rest of the parameters: training and test transformations, image size, and optimizer. See configs in Models Zoo. The trick was in heuristics in our miner and sampler:

- Category Balance Sampler forms the batches limiting the number of categories C in it. For instance, when $C = 1$ it puts only jackets in one batch and only jeans into another one (just an example). It automatically makes the negative pairs harder: it's more meaningful for a model to realise why two jackets are different than to understand the same about a jacket and a t-shirt.
- Hard Triplets Miner makes the task even harder keeping only the hardest triplets (with maximal positive and minimal negative distances).

Here are *CMC@1* scores for 2 popular benchmarks. SOP dataset: Hyp-ViT — 85.9, ours — 86.6. DeepFashion dataset: Hyp-ViT — 92.5, ours — 92.1. Thus, utilising simple heuristics and avoiding heavy math we are able to perform on SotA level.

Recent research in SSL definitely obtained great results. The problem is that these approaches required an enormous amount of computing to train the model. But in our framework, we consider the most common case when the average user has no more than a few GPUs.

At the same time, it would be unwise to ignore success in this sphere, so we still exploit it in two ways:

- As a source of checkpoints that would be great to start training with. From publications and our experience, they are much better as initialisation than the default supervised model trained on ImageNet. Thus, we added the

possibility to initialise your models using these pretrained checkpoints only by passing an argument in the config or the constructor.

- As a source of inspiration. For example, we adapted the idea of a memory bank from *MoCo* for the *TripletLoss*.

No, you don't. OML is a framework-agnostic. Despite we use PyTorch Lightning as a loop runner for the experiments, we also keep the possibility to run everything on pure PyTorch. Thus, only the tiny part of OML is Lightning-specific and we keep this logic separately from other code (see `oml.lightning`). Even when you use Lightning, you don't need to know it, since we provide ready to use Pipelines.

The possibility of using pure PyTorch and modular structure of the code leaves a room for utilizing OML with your favourite framework after the implementation of the necessary wrappers.

Yes. To run the experiment with Pipelines you only need to write a converter to our format (it means preparing the `.csv` table with 5 predefined columns). That's it!

Probably we already have a suitable pre-trained model for your domain in our *Models Zoo*. In this case, you don't even need to train it.

You can adapt OML to make it work not only with images. Just open one of the examples and replace `Dataset` remaining the rest of the pipeline the same or almost the same. There is several people who successfully used OML for texts in their real-world projects.

Unfortunately, we don't have ready-to-use tutorials for this kind of usage at the moment.

CHAPTER

THREE

CONTRIBUTING GUIDE

3.1 Before you start

- Read our FAQ.
- Check out python examples and Pipelines.

3.2 Choosing a task

- Check out our Kanban board. You can work on one of the existing issues or create a new one.
- Start the conversation under the issue you picked. We will discuss the design and content of the pull request, and then you can start working on it.

3.3 Contributing in general

- Fork the repository.
- Clone it locally.
- Create a branch with a name that speaks for itself.
- Set up the environment. You can install the library in dev mode via `pip install -e .` or build / pull docker image.
- Implement the discussed functionality, **docstrings**, and **tests** for it.
- Run tests locally using commands from `Makefile`.
- Push the code to your forked repository.
- Create a pull request to OpenMetricLearning.

3.4 Contributing to documentation

- If you want to change `README.md` you should go to `docs/readme`, change the desired section and then build readme via `make build_readme`. *So, don't change the main readme file directly, otherwise tests will fail.*
- Don't forget to update the documentation if needed. Its source is located in `docs/source`. To inspect it locally, you should run `make html` (from `docs` folder) and then open `docs/build/html/index.html` in your browser.

3.5 Contributing to models ZOO

- Add the model's implementation under `oml/models`.
- Implement `from_pretrained()` and add the corresponding transforms.
- Add the model to `oml/registry` and `oml/configs`.
- Evaluate model on 4 benchmarks and add the results into ZOO table in the main Readme.

3.6 Contributing to pipelines

- Implement your changes in one of the pipelines (`extractor_training_pipeline`, `extractor_validation_pipeline` or others).
- Add a new test or modify an existing one under `tests/test_runs/test_pipelines`.
- If adding a new test:
  - Add config file: `tests/test_runs/test_pipelines/configs/train_or_validate_new_feature.yaml`
  - Add python script: `tests/test_runs/test_pipelines/train_or_validate_new_feature.py`
  - Add test: `tests/test_runs/test_pipelines/test_pipelines.py`

3.7 Don't forget to update Registry

- If you want to add some new criterion, miner, model, optimizer, sampler, lr scheduler or transforms, don't forget to add it to the corresponding registry (see `oml.registry`) and also add a config file (see `oml.configs`).

CHAPTER

FOUR

DATASET FORMAT

To reuse as much from OML as possible, you need to prepare a *.csv* file in the required format. It's not obligatory, especially if you implement your own Dataset, but the format is required in case of usage built-in datasets or Pipelines. You can check out the tiny dataset as an example.

Required columns:

- `label` - integer value indicates the label of item.
- `path` - path to image. It may be global or relative path (in these case you need to pass `dataset_root` to build-in Datasets.)
- `split` - must be one of 2 values: `train` or `validation`.
- `is_query`, `is_gallery` - have to be `None` where `split == train` and `True` (or `1`) or `False` (or `0`) where `split == validation`. Note, that both values can be `True` at the same time. Then we will validate every item in the validation set using the "1 vs rest" approach (datasets of this kind are `SOP`, `CARS196` or `CUB`).

Optional columns:

- `category` - category which groups sets of similar labels (like `dresses`, or `furniture`).
- `x_1`, `x_2`, `y_1`, `y_2` - integers, the format is `left`, `right`, `top`, `bot` (`x_1` and `y_1` must be less than `x_2` and `y_2`). If only part of your images has bounding boxes, just fill the corresponding row with empty values.
- `sequence` - ids of sequences of photos that may be useful to handle in Re-id tasks. Must be strings or integers. Take a look at the detailed example.

Check out the examples of dataframes. You can also use helper to check if your dataset is in the right format:

```
import pandas as pd
from oml.utils.dataframe_format import check_retrieval_dataframe_format

check_retrieval_dataframe_format(df=pd.read_csv("/path/to/table.csv"), dataset_root="/
↪path/to/dataset/root/")
```

CHAPTER

FIVE

PIPELINES

This page is a good place to start and get an overview of Pipelines approach in general. For the details of the exact Pipeline, please, visit the corresponding page:

- features_extraction
- postprocessing (for results retrieved by feature extractor)

5.1 What are Pipelines?

Pipelines are a predefined collection of scripts/recipes that provide a way to run metric learning experiments by changing only the config. Pipelines require your data to be prepared in a special `.csv` file, described in details here.

Pipelines may help you if:

- You have a dataset which format can be aligned with one required by a pipeline
- You need reproducibility for your experiments
- You prefer changing config over diving into the code

They will not work if:

- You deal with a corner case and the flexibility of an existing Pipeline isn't enough

5.2 How to work with Pipelines?

The recommended way is the following:

1. Install OML: `pip install open-metric-learning`
2. Prepare your dataset in the required format. (There are converters for 4 popular datasets).
3. Go to Pipeline's folder and copy `.py` script and its `.yaml` to your workdir. Modify the config if needed.
4. Run the script via the command line.

5.3 Minimal example of a Pipeline

Each Pipeline is built around 3 components:

- Config file
- Registry of classes & functions
- Entrypoint function for a Pipeline + script to run it

Let's consider an oversimplified example: we create a model and "validate" it via applying it to a tensor of ones using the predefined device:

`pipeline.py` (implements the logic of a Pipeline, **part of OML package**):

```
import torch
from registry import get_model

def toy_validation(config):
  model = get_model(config["model"]).eval()
  inp = torch.ones((1, 3, 32, 32)).float()
  output = model(inp.to(config["device"]))
```

`registry.py` (maps entity's config to a Python constructor, **part of OML package**):

```
from torchvision.models import resnet18, resnet50

MODELS_REGISTRY = {"resnet18": resnet18, "resnet50": resnet50}

def get_model(config):
  return MODELS_REGISTRY[config["name"]](**config["args"])
```

`config.yaml` (describes the whole run, **your local file**):

```
model:
  name: resnet50
  args:
    weights: IMAGENET1K_V1

device: cpu
```

`validate.py` (script which simply runs pipeline, **your local file**):

```
import hydra
from pipeline import toy_validation

@hydra.main(config_name="config.yaml")
def main_hydra(cfg):
    toy_validation(cfg)

if __name__ == "__main__":
    main_hydra()
```

Shell command:

```
python validate.py model.args.weights=null
```

Note, we use Hydra as a config parser. One of its abilities is to change part of the config from a command line, as showed above.

5.4 Building blocks of Pipelines

Like every Python program Pipelines consist of functions and objects, that sum up in the desired logic. Some of them, like extractor or optimizer, may be completely replaced via config. Others, like a trainer or a metrics calculator will stay there anyway, but you can change their behaviour via config as well.

Let's say, you work with one of the Pipelines, and it assumes that an extractor must be a successor of IExtractor interface. You have two options if you want to use another extractor:

- You can check the existing successors of `IExtractor` in the library and pick one of them;
- Or you can implement your successor, see the section below for details.

To see what exact parts of each config can be modified, please, visit their subpages.

5.5 How to use my own implementation of loss, extractor, etc.?

You should put a constructor of your Python object inside the corresponding registry by some key. It allows you to access this object in the config file by that key.

Let's consider an example of using custom augmentations & extractor to train your feature extractor.

Your `train.py` and `config.yaml` may look like this:

```
import hydra
import torchvision.transforms as t
from omegaconf import DictConfig
from torchvision.models import resnet18

from oml.interfaces.models import IExtractor
from oml.lightning.pipelines.train import extractor_training_pipeline
from oml.registry.models import EXTRACTORS_REGISTRY
from oml.registry.transforms import TRANSFORMS_REGISTRY


class CustomExtractor(IExtractor):

  def __init__(self, pretrained):
    super().__init__()
    self.resnet = resnet18(pretrained=pretrained)

  def forward(self, x):
    return self.resnet(x)

  # this property is obligatory for IExtractor
  @property
  def feat_dim(self):
    return self.resnet.fc.out_features
```

(continues on next page)

(continued from previous page)

```
def get_custom_augs() -> t.Compose:
  return t.Compose([
    t.RandomHorizontalFlip(),
    t.RandomGrayscale(),
    t.ToTensor(),
    t.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]),
  ])


# Put extractor & transforms constructors to the registries
TRANSFORMS_REGISTRY["custom_augmentations"] = get_custom_augs
EXTRACTORS_REGISTRY["custom_extractor"] = CustomExtractor


@hydra.main(config_path="configs", config_name="train.yaml")
def main_hydra(cfg: DictConfig) -> None:
  extractor_training_pipeline(cfg)


if __name__ == "__main__":
  main_hydra()
```

```
...

transforms_train:
  name: custom_augmentations  # this name is a key for transforms registry we set above
  args: {}  # our augmentations have no obligatory initial arguments

extractor:
  name: custom_extractor  # this name is a key for models registry we set above
  args:
    pretrained: True  # our model has one argument that has to be set

...
```

The same logic works for optimisers, samplers, losses, etc., depending on the exact Pipeline and its building blocks.

5.6 Configuration via config is not flexible enough in my case

Let's say you want to change the implementation of `Dataset`, which is not configurable in a pipeline of your interest. In other words, you can only change its initial arguments, but cannot replace the corresponding class.

In this case, you can copy the source code of the main pipeline entrypoint function and modify it as you want. For example, if you want to train your feature extractor with your own implementation of `Dataset`, you need to copy & modify extractor_training_pipeline. To find an entrypoint function for other pipelines simply check what is used inside the desired `*.py` file.

CHAPTER

SIX

LOGGING & VISUALIZATION

6.1 Logging in Pipelines

There are several loggers integrated with Pipelines. You can also use your custom logger.

- Tensorboard — is active by default if there is no `logger` in config.

```
...
logger:
  name: tensorboard
  args:
    save_dir: "."
...
```

- Neptune

```
...
logger:
  name: neptune  # requires <NEPTUNE_API_TOKEN> as global env
  args:
    project: "oml-team/test"
...
```

```
export NEPTUNE_API_TOKEN=your_token; python train.py
```

- Weights and Biases

```
...
logger:
    name: wandb
    args:
      project: "test_project"
...
```

```
export WANDB_API_KEY=your_token; python train.py
```

- MLFlow

```
...
logger:
    name: mlflow
```

(continues on next page)

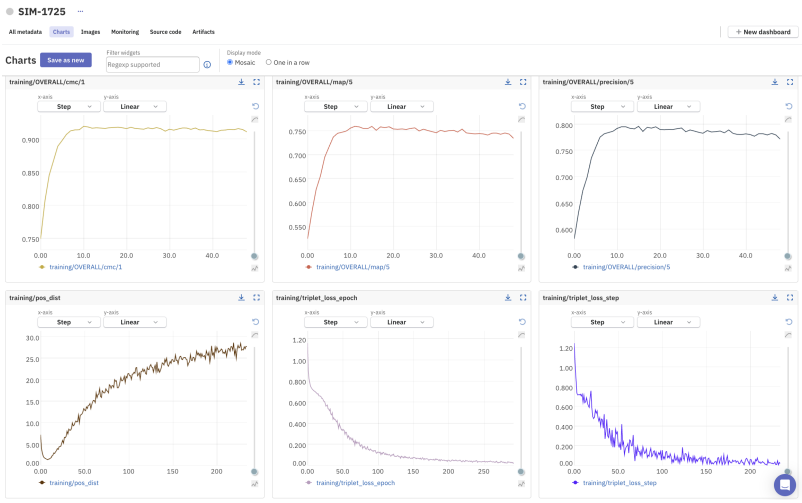
(continued from previous page)

```
    args:
        experiment_name: "test_project"
        tracking_uri: "file:./ml-runs"  # another way: export MLFLOW_TRACKING_
↪URI=file:./ml-runs
...
```

- ClearML

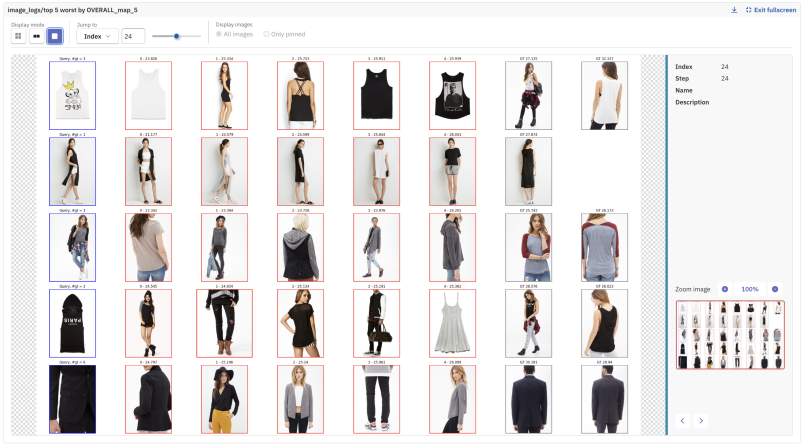
```
...
logger:
    name: clearml
    args:
        project_name: "test_project"
        task_name: "test"
        offline_mode: False # if True logging is directed to a local dir
...
```

An example of logging via Neptune in the feature extractor pipeline.



So, you get:

- Metrics such as `CMC@1`, `Precision@5`, `MAP@5`, which were provided in a config file as `metric_args`. Note, you can set `metrics_args.return_only_overall_category: False` to log metrics independently for each of the categories (if your dataset has ones).
- Loss values averaged over batches and epochs. Some of the built-in OML's losses have their unique additional statistics that is also logged. We used TripletLossWithMargin in our example, which comes along with tracking positive distances, negative distances and a fraction of active triplets (those for which loss is greater than zero).



The image above shows the worst model's predictions in terms of MAP@5 metric. In particular, each row contains:

- A query (blue)
- Five closest items from a gallery to the given query & the corresponding distances (they are all red because they are irrelevant to the query)
- At most two ground truths (grey), to get an idea of what model should return

You also get some artifacts for reproducibility, such as:

- Source code
- Config
- Dataframe
- Tags

6.2 Logging in Python

6.2.1 Using Lightning

Take a look at the following example: Training + Validation [Lightning and logging]. It shows how to use each of: Tensorboard, MLFlow, ClearML, Neptune or WandB.

6.2.2 Using plain Python

Log whatever information you want using the tool of your choice. We just provide some tips on how to get this information. There are two main sources of logs:

- Criterion (loss). Some of the built-in OML's losses have their unique additional statistics, which is stored in the `last_logs` field. See **Training** in the examples.
- Metrics calculator — EmbeddingMetrics. It has plenty of methods useful for logging. See **Validation** in the examples.

We also recommend you take a look at:

- Visualisation notebook for interactive errors analysis and visualizing attention maps.
- ViTExtractor.draw_attention()
- ResnetExtractor.draw_gradcam()

CHAPTER

SEVEN

EXAMPLES

Using Python-API is the most flexible approach: you are not limited by our project & config structures and you can use only the needed part of OML's functionality. You will find code snippets below to train, validate and inference the model on a tiny dataset of figures. Here are more details regarding dataset format.

Schemas, explanations and tips illustrating the code below.

```
import torch
from tqdm import tqdm

from oml.datasets.base import DatasetWithLabels
from oml.losses.triplet import TripletLossWithMiner
from oml.miners.inbatch_all_tri import AllTripletsMiner
from oml.models import ViTExtractor
from oml.samplers.balance import BalanceSampler
from oml.utils.download_mock_dataset import download_mock_dataset

dataset_root = "mock_dataset/"
df_train, _ = download_mock_dataset(dataset_root)

extractor = ViTExtractor("vits16_dino", arch="vits16", normalise_features=False).train()
optimizer = torch.optim.SGD(extractor.parameters(), lr=1e-6)

train_dataset = DatasetWithLabels(df_train, dataset_root=dataset_root)
criterion = TripletLossWithMiner(margin=0.1, miner=AllTripletsMiner(), need_logs=True)
sampler = BalanceSampler(train_dataset.get_labels(), n_labels=2, n_instances=2)
train_loader = torch.utils.data.DataLoader(train_dataset, batch_sampler=sampler)

for batch in tqdm(train_loader):
    embeddings = extractor(batch["input_tensors"])
    loss = criterion(embeddings, batch["labels"])
    loss.backward()
    optimizer.step()
    optimizer.zero_grad()

    # info for logging: positive/negative distances, number of active triplets
    print(criterion.last_logs)
```

```
import torch
from tqdm import tqdm
```

(continues on next page)

(continued from previous page)

```
from oml.datasets.base import DatasetQueryGallery
from oml.metrics.embeddings import EmbeddingMetrics
from oml.models import ViTExtractor
from oml.utils.download_mock_dataset import download_mock_dataset

dataset_root = "mock_dataset/"
_, df_val = download_mock_dataset(dataset_root)

extractor = ViTExtractor("vits16_dino", arch="vits16", normalise_features=False).eval()

val_dataset = DatasetQueryGallery(df_val, dataset_root=dataset_root)

val_loader = torch.utils.data.DataLoader(val_dataset, batch_size=4)
calculator = EmbeddingMetrics(extra_keys=("paths",))
calculator.setup(num_samples=len(val_dataset))

with torch.no_grad():
    for batch in tqdm(val_loader):
        batch["embeddings"] = extractor(batch["input_tensors"])
        calculator.update_data(batch)

metrics = calculator.compute_metrics()

# Logging
print(calculator.metrics)  # metrics
print(calculator.metrics_unreduced)  # metrics without averaging over queries

# Visualisation
calculator.get_plot_for_queries(query_ids=[0, 2], n_instances=5)  # draw predictions on␣
↪predefined queries
calculator.get_plot_for_worst_queries(metric_name="OVERALL/map/5", n_queries=2, n_
↪instances=5)  # draw mistakes
calculator.visualize()  # draw mistakes for all the available metrics
```

```
import torch

from oml.const import MOCK_DATASET_PATH
from oml.inference.flat import inference_on_images
from oml.models import ViTExtractor
from oml.registry.transforms import get_transforms_for_pretrained
from oml.utils.download_mock_dataset import download_mock_dataset
from oml.utils.misc_torch import pairwise_dist

_, df_val = download_mock_dataset(MOCK_DATASET_PATH)
df_val["path"] = df_val["path"].apply(lambda x: MOCK_DATASET_PATH / x)
queries = df_val[df_val["is_query"]]["path"].tolist()
galleries = df_val[df_val["is_gallery"]]["path"].tolist()

extractor = ViTExtractor.from_pretrained("vits16_dino")
```

(continues on next page)

(continued from previous page)

```
transform, _ = get_transforms_for_pretrained("vits16_dino")

args = {"num_workers": 0, "batch_size": 8}
features_queries = inference_on_images(extractor, paths=queries, transform=transform,␣
↪**args)
features_galleries = inference_on_images(extractor, paths=galleries, transform=transform,
↪ **args)

# Now we can explicitly build pairwise matrix of distances or save you RAM via using kNN
use_knn = False
top_k = 3

if use_knn:
    from sklearn.neighbors import NearestNeighbors
    knn = NearestNeighbors(algorithm="auto", p=2)
    knn.fit(features_galleries)
    dists, ii_closest = knn.kneighbors(features_queries, n_neighbors=top_k, return_
↪distance=True)

else:
    dist_mat = pairwise_dist(x1=features_queries, x2=features_galleries)
    dists, ii_closest = torch.topk(dist_mat, dim=1, k=top_k, largest=False)

print(f"Top {top_k} items closest to queries are:\n {ii_closest}")
```

```
import pytorch_lightning as pl
import torch

from oml.datasets.base import DatasetQueryGallery, DatasetWithLabels
from oml.lightning.modules.extractor import ExtractorModule
from oml.lightning.callbacks.metric import MetricValCallback
from oml.losses.triplet import TripletLossWithMiner
from oml.metrics.embeddings import EmbeddingMetrics
from oml.miners.inbatch_all_tri import AllTripletsMiner
from oml.models import ViTExtractor
from oml.samplers.balance import BalanceSampler
from oml.utils.download_mock_dataset import download_mock_dataset
from oml.lightning.pipelines.logging import (
    ClearMLPipelineLogger,
    MLFlowPipelineLogger,
    NeptunePipelineLogger,
    TensorBoardPipelineLogger,
    WandBPipelineLogger,
)

dataset_root = "mock_dataset/"
df_train, df_val = download_mock_dataset(dataset_root)

# model
extractor = ViTExtractor("vits16_dino", arch="vits16", normalise_features=False)
```

(continues on next page)

(continued from previous page)

```
# train
optimizer = torch.optim.SGD(extractor.parameters(), lr=1e-6)
train_dataset = DatasetWithLabels(df_train, dataset_root=dataset_root)
criterion = TripletLossWithMiner(margin=0.1, miner=AllTripletsMiner())
batch_sampler = BalanceSampler(train_dataset.get_labels(), n_labels=2, n_instances=3)
train_loader = torch.utils.data.DataLoader(train_dataset, batch_sampler=batch_sampler)

# val
val_dataset = DatasetQueryGallery(df_val, dataset_root=dataset_root)
val_loader = torch.utils.data.DataLoader(val_dataset, batch_size=4)
metric_callback = MetricValCallback(metric=EmbeddingMetrics(extra_keys=[train_dataset.
↪paths_key,]), log_images=True)

# 1) Logging with Tensorboard
logger = TensorBoardPipelineLogger(".")

# 2) Logging with Neptune
# logger = NeptunePipelineLogger(api_key="", project="", log_model_checkpoints=False)

# 3) Logging with Weights and Biases
# import os
# os.environ["WANDB_API_KEY"] = ""
# logger = WandBPipelineLogger(project="test_project", log_model=False)

# 4) Logging with MLFlow locally
# logger = MLFlowPipelineLogger(experiment_name="exp", tracking_uri="file:./ml-runs")

# 5) Logging with ClearML
# logger = ClearMLPipelineLogger(project_name="exp", task_name="test")

# run
pl_model = ExtractorModule(extractor, criterion, optimizer)
trainer = pl.Trainer(max_epochs=3, callbacks=[metric_callback], num_sanity_val_steps=0,␣
↪logger=logger)
trainer.fit(pl_model, train_dataloaders=train_loader, val_dataloaders=val_loader)
```

```
import pytorch_lightning as pl
import torch

from oml.datasets.base import DatasetQueryGallery, DatasetWithLabels
from oml.lightning.modules.extractor import ExtractorModuleDDP
from oml.lightning.callbacks.metric import MetricValCallbackDDP
from oml.losses.triplet import TripletLossWithMiner
from oml.metrics.embeddings import EmbeddingMetricsDDP
from oml.miners.inbatch_all_tri import AllTripletsMiner
from oml.models import ViTExtractor
from oml.samplers.balance import BalanceSampler
from oml.utils.download_mock_dataset import download_mock_dataset
from pytorch_lightning.strategies import DDPStrategy
```

(continues on next page)

(continued from previous page)

```
dataset_root = "mock_dataset/"
df_train, df_val = download_mock_dataset(dataset_root)

# model
extractor = ViTExtractor("vits16_dino", arch="vits16", normalise_features=False)

# train
optimizer = torch.optim.SGD(extractor.parameters(), lr=1e-6)
train_dataset = DatasetWithLabels(df_train, dataset_root=dataset_root)
criterion = TripletLossWithMiner(margin=0.1, miner=AllTripletsMiner())
batch_sampler = BalanceSampler(train_dataset.get_labels(), n_labels=2, n_instances=3)
train_loader = torch.utils.data.DataLoader(train_dataset, batch_sampler=batch_sampler)

# val
val_dataset = DatasetQueryGallery(df_val, dataset_root=dataset_root)
val_loader = torch.utils.data.DataLoader(val_dataset, batch_size=4)
metric_callback = MetricValCallbackDDP(metric=EmbeddingMetricsDDP())  # DDP specific

# run
pl_model = ExtractorModuleDDP(extractor=extractor, criterion=criterion,␣
↪optimizer=optimizer,
                              loaders_train=train_loader, loaders_val=val_loader  # DDP␣
↪specific
                              )

ddp_args = {"accelerator": "cpu", "devices": 2, "strategy": DDPStrategy(), "use_
↪distributed_sampler": False}  # DDP specific
trainer = pl.Trainer(max_epochs=1, callbacks=[metric_callback], num_sanity_val_steps=0,␣
↪**ddp_args)
trainer.fit(pl_model)  # we don't pass loaders to .fit() in DDP
```

Colab: there is no Colab link since it provides only single-GPU machines.

```
import pytorch_lightning as pl
import torch

from oml.datasets.base import DatasetQueryGallery
from oml.lightning.callbacks.metric import MetricValCallback
from oml.lightning.modules.extractor import ExtractorModule
from oml.metrics.embeddings import EmbeddingMetrics
from oml.models import ViTExtractor
from oml.transforms.images.torchvision import get_normalisation_resize_torch
from oml.utils.download_mock_dataset import download_mock_dataset

dataset_root = "mock_dataset/"
_, df_val = download_mock_dataset(dataset_root)

extractor = ViTExtractor("vits16_dino", arch="vits16", normalise_features=False)

# 1st validation dataset (big images)
val_dataset_1 = DatasetQueryGallery(df_val, dataset_root=dataset_root,
```

(continues on next page)

(continued from previous page)

```
                                    transform=get_normalisation_resize_torch(im_
↪size=224))
val_loader_1 = torch.utils.data.DataLoader(val_dataset_1, batch_size=4)
metric_callback_1 = MetricValCallback(metric=EmbeddingMetrics(extra_keys=[val_dataset_1.
↪paths_key,]),
                                      log_images=True, loader_idx=0)

# 2nd validation dataset (small images)
val_dataset_2 = DatasetQueryGallery(df_val, dataset_root=dataset_root,
                                    transform=get_normalisation_resize_torch(im_size=48))
val_loader_2 = torch.utils.data.DataLoader(val_dataset_2, batch_size=4)
metric_callback_2 = MetricValCallback(metric=EmbeddingMetrics(extra_keys=[val_dataset_2.
↪paths_key,]),
                                      log_images=True, loader_idx=1)

# run validation
pl_model = ExtractorModule(extractor, None, None)
trainer = pl.Trainer(max_epochs=3, callbacks=[metric_callback_1, metric_callback_2], num_
↪sanity_val_steps=0)
trainer.validate(pl_model, dataloaders=(val_loader_1, val_loader_2))

print(metric_callback_1.metric.metrics)
print(metric_callback_2.metric.metrics)
```

7.1 Usage with PyTorch Metric Learning

You can easily access a lot of content from PyTorch Metric Learning. The examples below are different from the basic ones only in a few lines of code:

```
import torch
from tqdm import tqdm

from oml.datasets.base import DatasetWithLabels
from oml.models import ViTExtractor
from oml.samplers.balance import BalanceSampler
from oml.utils.download_mock_dataset import download_mock_dataset

from pytorch_metric_learning import losses, distances, reducers, miners

dataset_root = "mock_dataset/"
df_train, _ = download_mock_dataset(dataset_root)

extractor = ViTExtractor("vits16_dino", arch="vits16", normalise_features=False).train()
optimizer = torch.optim.SGD(extractor.parameters(), lr=1e-6)

train_dataset = DatasetWithLabels(df_train, dataset_root=dataset_root)

# PML specific
# criterion = losses.TripletMarginLoss(margin=0.2, triplets_per_anchor="all")
criterion = losses.ArcFaceLoss(num_classes=df_train["label"].nunique(), embedding_
```

(continues on next page)

(continued from previous page)

```
↪size=extractor.feat_dim)  # for classification-like losses

sampler = BalanceSampler(train_dataset.get_labels(), n_labels=2, n_instances=2)
train_loader = torch.utils.data.DataLoader(train_dataset, batch_sampler=sampler)

for batch in tqdm(train_loader):
    embeddings = extractor(batch["input_tensors"])
    loss = criterion(embeddings, batch["labels"])
    loss.backward()
    optimizer.step()
    optimizer.zero_grad()
```

```
import torch
from tqdm import tqdm

from oml.datasets.base import DatasetWithLabels
from oml.models import ViTExtractor
from oml.samplers.balance import BalanceSampler
from oml.utils.download_mock_dataset import download_mock_dataset

from pytorch_metric_learning import losses, distances, reducers, miners

dataset_root = "mock_dataset/"
df_train, _ = download_mock_dataset(dataset_root)

extractor = ViTExtractor("vits16_dino", arch="vits16", normalise_features=False).train()
optimizer = torch.optim.SGD(extractor.parameters(), lr=1e-6)

train_dataset = DatasetWithLabels(df_train, dataset_root=dataset_root)

# PML specific
distance = distances.LpDistance(p=2)
reducer = reducers.ThresholdReducer(low=0)
criterion = losses.TripletMarginLoss()
miner = miners.TripletMarginMiner(margin=0.2, distance=distance, type_of_triplets="all")

sampler = BalanceSampler(train_dataset.get_labels(), n_labels=2, n_instances=2)
train_loader = torch.utils.data.DataLoader(train_dataset, batch_sampler=sampler)

for batch in tqdm(train_loader):
    embeddings = extractor(batch["input_tensors"])
    loss = criterion(embeddings, batch["labels"], miner(embeddings, batch["labels"]))  #␣
↪PML specific
    loss.backward()
    optimizer.step()
    optimizer.zero_grad()
```

To use content from PyTorch Metric Learning with our Pipelines just follow the standard tutorial of adding custom loss.

Note, during the validation process OpenMetricLearning computes *L2* distances. Thus, when choosing a distance from PML, we recommend you to pick *distances.LpDistance(p=2)*.

7.2 Handling sequences of photos

The below is mostly related to animal or person re-identification tasks, where observations are often done in the form of sequences of frames. The problem appears when calculating retrieval metrics, because the closest retrieved images most likely will be the neighbor frames from the same sequence as a query. Thus, we get good values of metrics. but don't really understand what is going on. So, it's better to ignore photos taken from the same sequence as a given query.

If you take a look at standard Re-id benchmarks as MARS dataset, you may see that ignoring frames from the same camera is a part of the actual protocol. Following the same logic, we introduced `sequence` field in our dataset format.

If sequence ids are provided, retrieved items having the same sequence id as a given query will be ignored.

Below is an example of how to label consecutive shoots of the tiger with the same `sequence`:

On the figure below we show how provided sequence labels affect metrics calculation:

metric	consider sequence?	value
CMC@1	no (top figure)	1.0
CMC@1	yes (bottom figure)	0.0
Precision@2	no (top figure)	0.5
Precision@2	yes (bottom figure)	0.5

To use this functionality you only need to provide `sequence` column in your dataframe (containing **strings** or **integers**) and pass `sequence_key` to `EmbeddingMetrics()`:

```
import torch
from tqdm import tqdm

from oml.datasets.base import DatasetQueryGallery
from oml.metrics.embeddings import EmbeddingMetrics
from oml.models import ViTExtractor
from oml.utils.download_mock_dataset import download_mock_dataset

dataset_root = "mock_dataset/"
_, df_val = download_mock_dataset(dataset_root, df_name="df_with_sequence.csv")  # <- sequence info is in the file

extractor = ViTExtractor("vits16_dino", arch="vits16", normalise_features=False).eval()

val_dataset = DatasetQueryGallery(df_val, dataset_root=dataset_root)

val_loader = torch.utils.data.DataLoader(val_dataset, batch_size=4)
calculator = EmbeddingMetrics(extra_keys=("paths",), sequence_key=val_dataset.sequence_key)
calculator.setup(num_samples=len(val_dataset))

with torch.no_grad():
    for batch in tqdm(val_loader):
        batch["embeddings"] = extractor(batch["input_tensors"])
        calculator.update_data(batch)
```

(continues on next page)

(continued from previous page)

```
metrics = calculator.compute_metrics()
```

CHAPTER

EIGHT

MODELS ZOO

Models, trained by us. The metrics below are for **224 x 224** images:

model	cmc1	dataset	weights	experiment
`ViTExtractor.from_pretrained("vits16_inshop")`	0.921	DeepFashion Inshop	link	link
`ViTExtractor.from_pretrained("vits16_sop")`	0.866	Stanford Online Products	link	link
`ViTExtractor.from_pretrained("vits16_cars")`	0.907	CARS 196	link	link
`ViTExtractor.from_pretrained("vits16_cub")`	0.837	CUB 200 2011	link	link

Models, trained by other researchers. Note, that some metrics on particular benchmarks are so high because they were part of the training dataset (for example `unicom`). The metrics below are for 224 x 224 images:

model	Stanford Online Products	DeepFashion InShop	CUB 200 2011	CARS 196
`ViTUnicomExtractor.from_pretrained("vitb16_unicom")`	0.700	0.734	0.847	0.916
`ViTUnicomExtractor.from_pretrained("vitb32_unicom")`	0.690	0.722	0.796	0.893
`ViTUnicomExtractor.from_pretrained("vitl14_unicom")`	0.726	0.790	0.868	0.922
`ViTUnicomExtractor.from_pretrained("vitl14_336px_unicom")`	0.745	0.810	0.875	0.924
`ViTCLIPExtractor.from_pretrained("sber_vitb32_224")`	0.547	0.514	0.448	0.618
`ViTCLIPExtractor.from_pretrained("sber_vitb16_224")`	0.565	0.565	0.524	0.648
`ViTCLIPExtractor.from_pretrained("sber_vitl14_224")`	0.512	0.555	0.606	0.707
`ViTCLIPExtractor.from_pretrained("openai_vitb32_224")`	0.612	0.491	0.560	0.693
`ViTCLIPExtractor.from_pretrained("openai_vitb16_224")`	0.648	0.606	0.665	0.767
`ViTCLIPExtractor.from_pretrained("openai_vitl14_224")`	0.670	0.675	0.745	0.844
`ViTExtractor.from_pretrained("vits16_dino")`	0.648	0.509	0.627	0.265
`ViTExtractor.from_pretrained("vits8_dino")`	0.651	0.524	0.661	0.315
`ViTExtractor.from_pretrained("vitb16_dino")`	0.658	0.514	0.541	0.288
`ViTExtractor.from_pretrained("vitb8_dino")`	0.689	0.599	0.506	0.313
`ViTExtractor.from_pretrained("vits14_dinov2")`	0.566	0.334	0.797	0.503
`ViTExtractor.from_pretrained("vits14_reg_dinov2")`	0.566	0.332	0.795	0.740
`ViTExtractor.from_pretrained("vitb14_dinov2")`	0.565	0.342	0.842	0.644
`ViTExtractor.from_pretrained("vitb14_reg_dinov2")`	0.557	0.324	0.833	0.828
`ViTExtractor.from_pretrained("vitl14_dinov2")`	0.576	0.352	0.844	0.692
`ViTExtractor.from_pretrained("vitl14_reg_dinov2")`	0.571	0.340	0.840	0.871
`ResnetExtractor.from_pretrained("resnet50_moco_v2")`	0.493	0.267	0.264	0.149
`ResnetExtractor.from_pretrained("resnet50_imagenet1k_v1")`	0.515	0.284	0.455	0.247

**The metrics may be different from the ones reported by papers, because the version of train/val split and usage of bounding boxes may differ.*

8.1 How to use models from Zoo?

```python
from oml.const import CKPT_SAVE_ROOT as CKPT_DIR, MOCK_DATASET_PATH as DATA_DIR
from oml.models import ViTExtractor
from oml.registry.transforms import get_transforms_for_pretrained

model = ViTExtractor.from_pretrained("vits16_dino")
transforms, im_reader = get_transforms_for_pretrained("vits16_dino")

img = im_reader(DATA_DIR / "images" / "circle_1.jpg")  # put path to your image here
img_tensor = transforms(img)
# img_tensor = transforms(image=img)["image"]  # for transforms from Albumentations

features = model(img_tensor.unsqueeze(0))

# Check other available models:
print(list(ViTExtractor.pretrained_models.keys()))

# Load checkpoint saved on a disk:
model_ = ViTExtractor(weights=CKPT_DIR / "vits16_dino.ckpt", arch="vits16", normalise_
↪features=False)
```

We recommend you to open this page on GitHub, so you can see the corresponding scripts and configs.

CHAPTER

NINE

PIPELINES: FEATURES EXTRACTION

- What are pipelines?
- Introduction to metric learning:

English | Russian | Chinese

These particular pipelines allow you to train, validate and inference models that represent images as feature vectors. In this section we explain how the following pipelines work under the hood:

- extractor_training_pipeline including training + validation
- extractor_validation_pipeline including validation only
- extractor_prediction_pipeline including saving extracted features

Pipelines also have the corresponding analogues in plain Python.

9.1 Training

It is expected that the dataset will be in the desired format. You can see a tiny figures dataset as an example.

To get used to terminology you can check the Glossary (naming convention).

This pipeline support two types of losses:

- Contrastive ones, like TripletLoss. They require special Miner and Batches Sampler. Miner produces triplets exploiting different strategies like hard mining, in its turn Sampler guarantees that batch contains enough different labels to form at least one triplet so miner can do its job.
- Classification ones, like ArcFace. They have no mining step by design and batch sampling strategy is optional for them. For these losses we consider the output of the layer before the classification head (which is a part of criterion in our implementation) as a feature vector.

Note! Despite the different nature of the losses above, they share the same forward signature: `forward(features, labels)`. That is why mining is happening inside the forward pass, see TripletLossWithMiner.

9.2 Validation

Validation part consists of the following steps:

1. Accumulating all the embeddings in EmbeddingMetrics.
2. Calculating distances between queries and galleries.
3. [Optional] Applying some specific retrieval postprocessing techniques like re-ranking.
4. Calculating retrieval metrics like CMC@k, Precision@k, MeanAveragePrecision@k or others.

9.3 Prediction / Inference

Prediction pipeline runs inference of a trained model and saves extracted features to the disk. Note, to speed up inference you can easily turn on multi GPU setup in the corresponding config file.

9.4 Customization

Pipelines are built around blocks like model, criterion, optimizer and so on. Some of them can be replaced by existing entities from OML or by your custom implementations, see the customisation instruction.

In feature extraction pipelines you can customize:

Block in config	Registry*	Example configs	Requirements on custom implementation
`transforms`	`TRANSFORMS_REGISTRY`	configs	Callable, see available.
`transforms_val`	`TRANSFORMS_REGISTRY`	configs	Callable, see available.
`extractor`	`EXTRACTORS_REGISTRY`	configs	Any successor of IExtractor, see available.
`sampler`	`SAMPLERS_REGISTRY`	configs	For losses with mining see this. For classification losses: no restrictions, but set `null` for RandomSampler.
`criterion`	`LOSSES_REGISTRY`	configs	The signature is required: `forward(features, labels)`. For contrastive losses: mining is implemented inside the forward pass. For classification losses: a classification head is a part of criterion. See available.
`optimizer`	`OPTIMIZERS_REGISTRY`	configs	Any regular PyTorch optimizer.
`scheduling`	`SCHEDULERS_REGISTRY`	configs	Any regular PyTorch lr scheduler, structured in Lightning format.
`logger`	`LOGGERS_REGISTRY`	configs	Child of IPipelineLogger

*Use: `from oml.registry import X_REGISTRY`.

9.5 Tips

We left plenty of comments in the training config for the CARS dataset, so you can start checking it out.

- If you don't know what parameters to pick for BalanceSampler, simply set `n_labels` equal to the median size of your classes, and set `n_instances` as big as your GPU allows for the given `n_labels`.
- Tips for TripletLossWithMiner:
  - The margin value of `0.2` may be a good choice if your extractor produces normalised features.
  - Triplet loss may struggle with a mode collapse: the situation when your loss goes down, but then fluctuates on a plateau on the level of margin value, which means that positive and negative distances both equal to zero. In this case, you can try to use the soft version of triplet loss instead (just set `margin: null`). You can also switch between mining strategies (*hard* / *all*).
  - Don't use `margin: null` if you normalise features since it breaks gradients flow (it can be proved mathematically).
- Check out Logging & Visualization to learn more about built-in possibilities such as tracking metrics, losses, geometric statistics over embeddings, visual inspection of a model's predictions and so on.

CHAPTER

TEN

EXAMPLES

You can also boost retrieval accuracy of your features extractor by adding a postprocessor (we recommend to check the examples above first). In the example below we train a siamese model to re-rank top retrieval outputs of the original model by performing inference on pairs `(query, output_i)` where `i=1..top_n`.

```
from pprint import pprint

import torch
from torch.nn import BCEWithLogitsLoss
from torch.utils.data import DataLoader

from oml.datasets.base import DatasetWithLabels, DatasetQueryGallery
from oml.inference.flat import inference_on_dataframe
from oml.metrics.embeddings import EmbeddingMetrics
from oml.miners.pairs import PairsMiner
from oml.models import ConcatSiamese, ViTExtractor
from oml.retrieval.postprocessors.pairwise import PairwiseImagesPostprocessor
from oml.samplers.balance import BalanceSampler
from oml.transforms.images.torchvision import get_normalisation_resize_torch
from oml.utils.download_mock_dataset import download_mock_dataset

# Let's start with saving embeddings of a pretrained extractor for which we want to build␣
↪a postprocessor
dataset_root = "mock_dataset/"
download_mock_dataset(dataset_root)

extractor = ViTExtractor("vits16_dino", arch="vits16", normalise_features=False)
transform = get_normalisation_resize_torch(im_size=64)

embeddings_train, embeddings_val, df_train, df_val = \
    inference_on_dataframe(dataset_root, "df.csv", extractor=extractor,␣
↪transforms=transform)

# We are building Siamese model on top of existing weights and train it to recognize␣
↪positive/negative pairs
siamese = ConcatSiamese(extractor=extractor, mlp_hidden_dims=[100])
optimizer = torch.optim.SGD(siamese.parameters(), lr=1e-6)
miner = PairsMiner(hard_mining=True)
criterion = BCEWithLogitsLoss()

train_dataset = DatasetWithLabels(df=df_train, transform=transform, extra_data={
```

(continues on next page)

(continued from previous page)

```
↪"embeddings": embeddings_train})
batch_sampler = BalanceSampler(train_dataset.get_labels(), n_labels=2, n_instances=2)
train_loader = DataLoader(train_dataset, batch_sampler=batch_sampler)

for batch in train_loader:
    # We sample pairs on which the original model struggled most
    ids1, ids2, is_negative_pair = miner.sample(features=batch["embeddings"],␣
↪labels=batch["labels"])
    probs = siamese(x1=batch["input_tensors"][ids1], x2=batch["input_tensors"][ids2])
    loss = criterion(probs, is_negative_pair.float())

    loss.backward()
    optimizer.step()
    optimizer.zero_grad()

# Siamese re-ranks top-n retrieval outputs of the original model performing inference on␣
↪pairs (query, output_i)
val_dataset = DatasetQueryGallery(df=df_val, extra_data={"embeddings": embeddings_val},␣
↪transform=transform)
valid_loader = DataLoader(val_dataset, batch_size=4, shuffle=False)

postprocessor = PairwiseImagesPostprocessor(top_n=3, pairwise_model=siamese,␣
↪transforms=transform)
calculator = EmbeddingMetrics(postprocessor=postprocessor)
calculator.setup(num_samples=len(val_dataset))

for batch in valid_loader:
    calculator.update_data(data_dict=batch)

pprint(calculator.compute_metrics())  # Pairwise inference happens here
```

```
import torch
from torch.utils.data import DataLoader

from oml.const import PATHS_COLUMN
from oml.datasets.base import DatasetQueryGallery
from oml.inference.flat import inference_on_dataframe
from oml.models import ConcatSiamese, ViTExtractor
from oml.registry.transforms import get_transforms_for_pretrained
from oml.retrieval.postprocessors.pairwise import PairwiseImagesPostprocessor
from oml.utils.download_mock_dataset import download_mock_dataset
from oml.utils.misc_torch import pairwise_dist

dataset_root = "mock_dataset/"
download_mock_dataset(dataset_root)

# 1. Let's use feature extractor to get predictions
extractor = ViTExtractor.from_pretrained("vits16_dino")
transforms, _ = get_transforms_for_pretrained("vits16_dino")
```

(continues on next page)

(continued from previous page)

```
_, emb_val, _, df_val = inference_on_dataframe(dataset_root, "df.csv", extractor,␣
↪transforms=transforms)

is_query = df_val["is_query"].astype('bool').values
distances = pairwise_dist(x1=emb_val[is_query], x2=emb_val[~is_query])

print("\nOriginal predictions:\n", torch.topk(distances, dim=1, k=3, largest=False)[1])

# 2. Let's initialise a random pairwise postprocessor to perform re-ranking
siamese = ConcatSiamese(extractor=extractor, mlp_hidden_dims=[100])  # Note! Replace it␣
↪with your trained postprocessor
postprocessor = PairwiseImagesPostprocessor(top_n=3, pairwise_model=siamese,␣
↪transforms=transforms)

dataset = DatasetQueryGallery(df_val, extra_data={"embeddings": emb_val},␣
↪transform=transforms)
loader = DataLoader(dataset, batch_size=4)

query_paths = df_val[PATHS_COLUMN][is_query].values
gallery_paths = df_val[PATHS_COLUMN][~is_query].values
distances_upd = postprocessor.process(distances=distances, queries=query_paths,␣
↪galleries=gallery_paths)

print("\nPredictions after postprocessing:\n", torch.topk(distances_upd, dim=1, k=3,␣
↪largest=False)[1])
```

The documentation for related classes is available via the link.

You can also check the corresponding pipeline analogue.

CHAPTER

ELEVEN

MODELS ZOO

There is no Zoo yet, but here you can download checkpoints manually.

We recommend you to open this page on GitHub*, so you can see the corresponding scripts and configs.*

CHAPTER

TWELVE

PAIRWISE POSTPROCESSING (RE-RANKING)

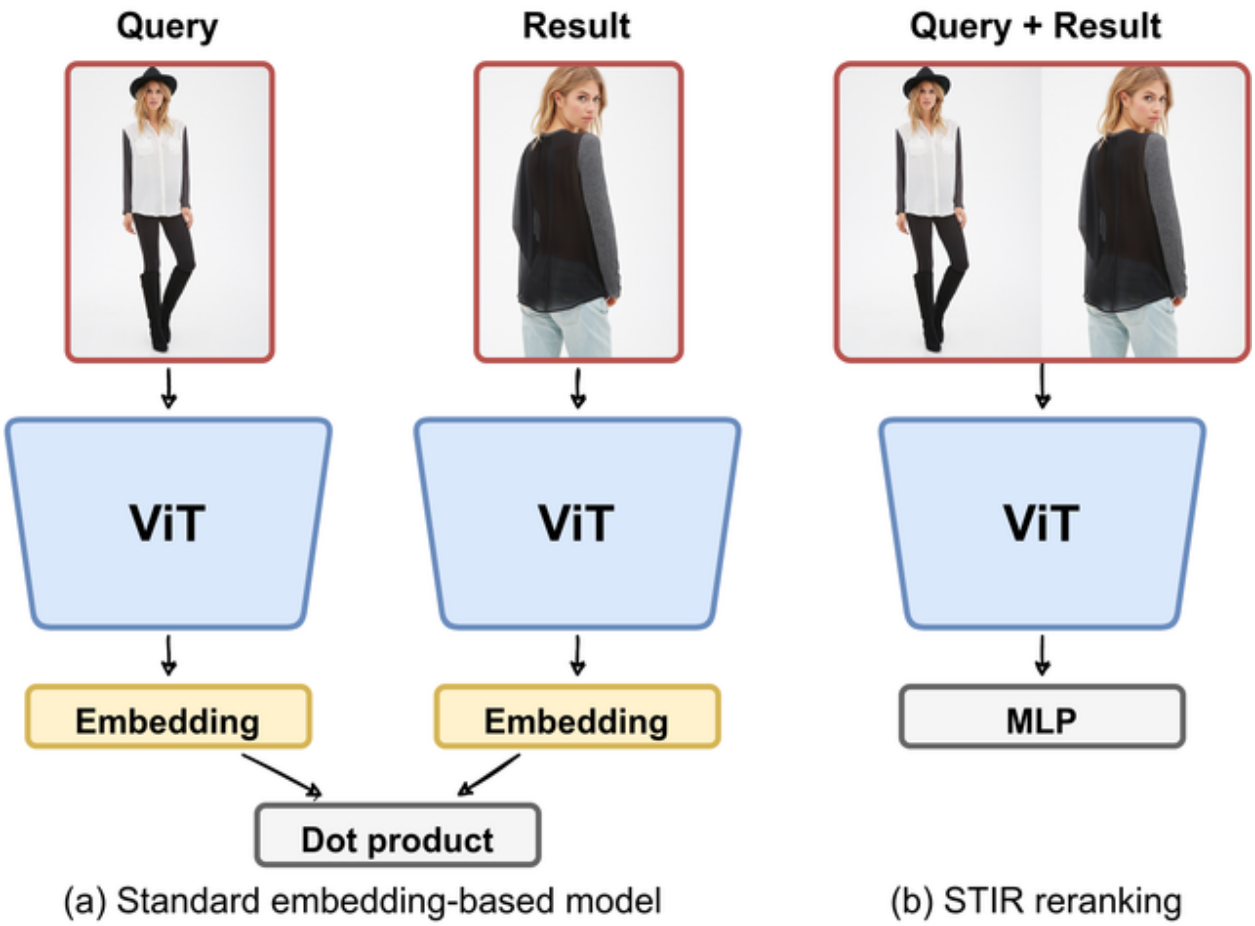
If you are new to Pipelines idea, you can start with general overview.

This Pipeline is based on the following study, completed by the OML's team:

12.1 STIR: Siamese Transformer for Image Retrieval Postprocessing

In this work, we first construct a baseline model trained with triplet loss with hard negatives mining that performs at the state of the art level but remains simple. Second, we introduce a novel approach for image retrieval postprocessing called Siamese Transformer for Image Retrieval (STIR) that re-ranks several top outputs in a single forward pass. Unlike previously proposed Reranking Transformers, STIR does not rely on global/local feature extraction and directly compares a query image and a retrieved candidate on pixel level with the usage of attention mechanism. The resulting approach defines a new state of the art on standard image retrieval datasets: Stanford Online Products and DeepFashion In-shop.

OPEN INTERACTIVE DEMO | demo's repository



12.1.1 I. Train & validate a feature extractor

DeepFashion Inshop

```
python train_extractor.py dataset_root=data/InShop/ logs_root=logs/InShop
python validate_extractor.py dataset_root=data/Inshop/ weights=extractor_inshop.ckpt
```

Stanford Online Products

```
python train_extractor.py dataset_root=data/SOP/ logs_root=logs/SOP
python validate_extractor.py dataset_root=data/SOP/ weights=extractor_sop.ckpt
```

12.1.2 II. Train & validate a postprocessor

DeepFashion Inshop

```
python train_postprocessor.py dataset_root=data/InShop/ logs_root=logs/InShop extractor_
↪weights=extractor_inshop.ckpt
python validate_postprocessor.py dataset_root=data/InShop/ extractor_weights=extractor_
↪inshop.ckpt postprocessor_weights=postprocessor_inshop.ckpt
```

Stanford Online Products

```
python train_postprocessor.py dataset_root=data/SOP/ logs_root=logs/SOP extractor_
↪weights=extractor_sop.ckpt
python validate_postprocessor.py dataset_root=data/SOP/ extractor_weights=extractor_sop.
↪ckpt postprocessor_weights=postprocessor_sop.ckpt
```

12.1.3 Pretrained checkpoints

If you don't want to perform training by yourself, you can download all the checkpoints mentioned above here, namely:

- `extractor_inshop.ckpt`
- `extractor_sop.ckpt`
- `postprocessor_inshop.ckpt`
- `postprocessor_sop.ckpt`

CHAPTER

THIRTEEN

BASE INTERFACES

13.1 IExtractor

class oml.interfaces.models.**IExtractor**(**args*, ***kwargs*)

Bases: Module, ABC

Models have to inherit this interface to be comparable with the rest of the library.

property feat_dim: int

The only method that obligatory to implemented.

extract(*x: Tensor*) → Tensor

classmethod from_pretrained(*weights: str*) → *IExtractor*

This method allows to download a pretrained checkpoint. The class field `self.pretrained_models` is the dictionary which keeps records of all the available checkpoints in the format, depending on implementation of a particular child of `IExtractor`. As a user, you don't need to worry about implementing this method.

Parameters

weights – A unique identifier (key) of a pretrained model information stored in a class field `self.pretrained_models`.

Returns: An instance of `IExtractor`

13.2 IPairwiseModel

class oml.interfaces.models.**IPairwiseModel**(**args*, ***kwargs*)

Bases: `Module`

A model of this type takes two inputs, for example, two embeddings or two images.

forward(*x1: Any*, *x2: Any*) → Tensor

Parameters

- **x1** – The first input.
- **x2** – The second input.

predict(*x1: Any*, *x2: Any*) → Tensor

While `self.forward()` is called during training, this method is called during inference or validation time. For example, it allows application of some activation, which was a part of a loss function during the training.

Parameters

- **x1** – The first input.
- **x2** – The second input.

13.3 IFreezable

class oml.interfaces.models.**IFreezable**

Bases: `ABC`

Models which can freeze and unfreeze their parts.

freeze() → None

Function for freezing. You can use it to partially freeze a model.

unfreeze() → None

Function for unfreezing. You can use it to unfreeze a model.

13.4 IBatchSampler

class oml.interfaces.samplers.**IBatchSampler**

Bases: `ABC`

We introduce our interface instead of using the default BatchSampler from Torch, because the last one is just a wrapper for the sequential sampler, which is not convenient for our purposes.

abstract __len__() → int

Returns
The number of batches in an epoch

abstract __iter__() → Iterator[List[int]]

Returns
Iterator contains indices for the batches

13.5 ITripletLossWithMiner

class oml.interfaces.criterions.**ITripletLossWithMiner**(**args*, ***kwargs*)

Bases: Module

Base class for TripletLoss combined with Miner.

forward(*features: Tensor*, *labels: Union[Tensor, List[int]]*) → Tensor

Parameters

- **features** – Features with the shape [batch_size, features_dim]
- **labels** – Labels with the size of batch_size

Returns
Loss value

13.6 IBaseDataset

class oml.interfaces.datasets.**IBaseDataset**(**args*, ***kwds*)

Bases: Dataset

13.7 ILabeledDataset

class oml.interfaces.datasets.**ILabeledDataset**(**args*, ***kwds*)

Bases: *IBaseDataset*, ABC

This is an interface for the datasets which provide labels of containing items.

__getitem__(*item: int*) → Dict[str, Any]

Parameters
item – Idx of the sample

Return type

Dictionary including the following keys

self.labels_key

abstract get_labels() → ndarray

13.8 IQueryGalleryDataset

class oml.interfaces.datasets.**IQueryGalleryDataset**(**args*, ***kwds*)

Bases: *IBaseDataset*, ABC

This is an interface for the datasets which hold the information on how to split the data into the query and gallery. The query and gallery ids may overlap. It doesn't need the ground truth labels, so it can be used for prediction on not annotated data.

abstract get_query_ids() → LongTensor

abstract get_gallery_ids() → LongTensor

13.9 IQueryGalleryLabeledDataset

class oml.interfaces.datasets.**IQueryGalleryLabeledDataset**(**args*, ***kwds*)

Bases: *IQueryGalleryDataset*, *ILabeledDataset*, ABC

This interface is similar to *IQueryGalleryDataset*, but there are ground truth labels.

abstract get_query_ids() → LongTensor

abstract get_gallery_ids() → LongTensor

abstract get_labels() → ndarray

13.10 IPairsDataset

class oml.interfaces.datasets.**IPairsDataset**(**args*, ***kwds*)

Bases: Dataset, ABC

This is an interface for the datasets which return pair of something.

__init__()

abstract __getitem__(*item: int*) → Dict[str, Any]

Parameters
item – Idx of the sample

Return type

Dictionary with the following keys

`self.pairs_1st_key` `self.pairs_2nd_key` `self.index_key`

13.11 IVisualizableDataset

class oml.interfaces.datasets.**IVisualizableDataset**(**args*, ***kwds*)

Bases: Dataset, ABC

Base class for the datasets which know how to visualise their items.

abstract visualize(*item: int*, *color: Tuple[int, int, int]*) → ndarray

13.12 IBasicMetric

class oml.interfaces.metrics.**IBasicMetric**

Bases: ABC

This is a base interface for the objects that calculate metrics.

abstract setup(**args: Any*, ***kwargs: Any*) → Any

Method for preparing metrics to work: memory allocation, placeholder preparation, etc. Has to be called before the first call of `self.update_data()`.

abstract update_data(**args: Any*, ***kwargs: Any*) → Any

Method for passing data to calculate the metrics later on.

abstract compute_metrics(**args: Any*, ***kwargs: Any*) → Any

The output must be in the following format:

```
{
    "self.overall_categories_key": {"metric1": ..., "metric2": ...},
    "category1": {"metric1": ..., "metric2": ...},
    "category2": {"metric1": ..., "metric2": ...}
}
```

Where `category1` and `category2` are optional.

13.13 ITripletsMiner

class oml.interfaces.miners.**ITripletsMiner**

Bases: ABC

An abstraction of triplet miner.

abstract sample(*features: Tensor*, *labels: Union[List[int], Tensor]*) → Tuple[Tensor, Tensor, Tensor]

This method includes the logic of mining/sampling triplets.

Parameters

- **features** – Features with the shape of `[batch_size, feature_size]`
- **labels** – Labels with the size of `batch_size`

Returns

Batch of triplets

13.14 ITripletsMinerInBatch

class oml.interfaces.miners.**ITripletsMinerInBatch**

Bases: *ITripletsMiner*

We expect that the child instances of this class will be used for mining triplets inside the batches. The batches must contain at least 2 samples for each class and at least 2 different labels, such behaviour can be guarantee via using samplers from our registry.

But you are not limited to using it in any other way.

abstract _sample(*features: Tensor*, *labels: List[int]*) → Tuple[List[int], List[int], List[int]]

This method includes the logic of mining triplets inside the batch. It can be based on information about the distance between the features, or the choice can be performed randomly.

Parameters

- **features** – Features with the shape of `[batch_size, feature_size]`
- **labels** – Labels with the size of `batch_size`

Returns
Indices of the batch samples to form the triplets

sample(*features: Tensor*, *labels: Union[List[int], Tensor]*) → Tuple[Tensor, Tensor, Tensor]

Parameters

- **features** – Features with the shape of `[batch_size, feature_size]`
- **labels** – Labels with the size of `batch_size`

Returns
Batch of triplets

13.15 IPipelineLogger

class oml.interfaces.loggers.**IPipelineLogger**

Bases: `Logger`, `IFigureLogger`

abstract log_figure(*fig: Figure*, *title: str*, *idx: int*) → None

abstract log_pipeline_info(*cfg: Union[Dict[str, Any], DictConfig]*) → None

CHAPTER

FOURTEEN

DATASETS

14.1 ImageBaseDataset

class oml.datasets.images.**ImageBaseDataset**(*paths: List[Path], dataset_root: Optional[Union[Path, str]] = None, bboxes: Optional[Sequence[Optional[Tuple[int, int, int, int]]]] = None, extra_data: Optional[Dict[str, Any]] = None, transform: Optional[Union[Compose, Compose]] = None, f_imread: Optional[Callable[[Union[Path, str, bytes]], Union[Image, ndarray]]] = None, cache_size: Optional[int] = 0, input_tensors_key: str = 'input_tensors', index_key: str = 'idx', paths_key: str = 'paths', x1_key: str = 'x1', x2_key: str = 'x2', y1_key: str = 'y1', y2_key: str = 'y2'*)

Bases: `IBaseDataset`, `IVisualizableDataset`

The base class that handles image specific logic.

__init__(*paths: List[Path], dataset_root: Optional[Union[Path, str]] = None, bboxes: Optional[Sequence[Optional[Tuple[int, int, int, int]]]] = None, extra_data: Optional[Dict[str, Any]] = None, transform: Optional[Union[Compose, Compose]] = None, f_imread: Optional[Callable[[Union[Path, str, bytes]], Union[Image, ndarray]]] = None, cache_size: Optional[int] = 0, input_tensors_key: str = 'input_tensors', index_key: str = 'idx', paths_key: str = 'paths', x1_key: str = 'x1', x2_key: str = 'x2', y1_key: str = 'y1', y2_key: str = 'y2'*)

Parameters

- **paths** – Paths to images. Will be concatenated with `dataset_root` if provided.
- **dataset_root** – Path to the images' dir, set `None` if you provided the absolute paths in your dataframe
- **bboxes** – Bounding boxes of images. Some of the images may not have bounding bboxes.

- **extra_data** – Dictionary containing records of some additional information.
- **transform** – Augmentations for the images, set `None` to perform only normalisation and casting to tensor
- **f_imread** – Function to read the images, pass `None` to pick it automatically based on provided transforms
- **cache_size** – Size of the dataset's cache
- **input_tensors_key** – Key to put tensors into the batches
- **index_key** – Key to put samples' ids into the batches
- **paths_key** – Key put paths into the batches # todo 522: remove
- **x1_key** – Key to put `x1` into the batches # todo 522: remove
- **x2_key** – Key to put `x2` into the batches # todo 522: remove
- **y1_key** – Key to put `y1` into the batches # todo 522: remove
- **y2_key** – Key to put `y2` into the batches # todo 522: remove

__getitem__(*item: int*) → Dict[str, Union[FloatTensor, int]]

Parameters
item – Idx of the sample

Returns
`self.input_tensors_key self.index_key: int = item`

Return type
Dictionary including the following keys

visualize(*item: int*, *color: Tuple[int, int, int] = (0, 0, 0)*) → ndarray

14.2 ImageLabeledDataset

class oml.datasets.images.**ImageLabeledDataset**(*df: DataFrame*, *extra_data: Optional[Dict[str, Any]] = None*, *dataset_root: Optional[Union[Path, str]] = None*, *transform: Optional[Compose] = None*, *f_imread: Optional[Callable[[Union[Path, str, bytes]], Union[Image, ndarray]]] = None*, *cache_size: Optional[int] = 0*, *input_tensors_key: str = 'input_tensors'*, *labels_key: str = 'labels'*, *index_key: str = 'idx'*, *paths_key: str = 'paths'*, *categories_key: Optional[str] = 'categories'*, *sequence_key: Optional[str] = 'sequence'*, *x1_key: str = 'x1'*, *x2_key: str = 'x2'*, *y1_key: str = 'y1'*, *y2_key: str = 'y2'*)

Bases: *ImageBaseDataset*, *ILabeledDataset*

The dataset of images having their ground truth labels.

__init__(*df: DataFrame*, *extra_data: Optional[Dict[str, Any]] = None*, *dataset_root: Optional[Union[Path, str]] = None*, *transform: Optional[Compose] = None*, *f_imread: Optional[Callable[[Union[Path, str, bytes]], Union[Image, ndarray]]] = None*, *cache_size: Optional[int] = 0*, *input_tensors_key: str = 'input_tensors'*, *labels_key: str = 'labels'*, *index_key: str = 'idx'*, *paths_key: str = 'paths'*, *categories_key: Optional[str] = 'categories'*, *sequence_key: Optional[str] = 'sequence'*, *x1_key: str = 'x1'*, *x2_key: str = 'x2'*, *y1_key: str = 'y1'*, *y2_key: str = 'y2'*)

Parameters

- **paths** – Paths to images. Will be concatenated with `dataset_root` if provided.
- **dataset_root** – Path to the images' dir, set `None` if you provided the absolute paths in your dataframe
- **bboxes** – Bounding boxes of images. Some of the images may not have bounding bboxes.
- **extra_data** – Dictionary containing records of some additional information.
- **transform** – Augmentations for the images, set `None` to perform only normalisation and casting to tensor
- **f_imread** – Function to read the images, pass `None` to pick it automatically based on provided transforms
- **cache_size** – Size of the dataset's cache
- **input_tensors_key** – Key to put tensors into the batches
- **index_key** – Key to put samples' ids into the batches
- **paths_key** – Key put paths into the batches # todo 522: remove
- **x1_key** – Key to put `x1` into the batches # todo 522: remove
- **x2_key** – Key to put `x2` into the batches # todo 522: remove
- **y1_key** – Key to put `y1` into the batches # todo 522: remove
- **y2_key** – Key to put `y2` into the batches # todo 522: remove

__getitem__(*item: int*) → Dict[str, Any]

Parameters
item – Idx of the sample

Return type

Dictionary including the following keys

`self.labels_key`

get_labels() → ndarray

visualize(*item: int*, *color: Tuple[int, int, int] = (0, 0, 0)*) → ndarray

14.3 ImageQueryGalleryLabeledDataset

class oml.datasets.images.**ImageQueryGalleryLabeledDataset**(*df: DataFrame*, *extra_data: Optional[Dict[str, Any]] = None*, *dataset_root: Optional[Union[Path, str]] = None*, *transform: Optional[Compose] = None*, *f_imread: Optional[Callable[[Union[Path, str, bytes]], Union[Image, ndarray]]] = None*, *cache_size: Optional[int] = 0*, *input_tensors_key: str = 'input_tensors'*, *labels_key: str = 'labels'*, *paths_key: str = 'paths'*, *categories_key: Optional[str] = 'categories'*, *sequence_key: Optional[str] = 'sequence'*, *x1_key: str = 'x1'*, *x2_key: str = 'x2'*, *y1_key: str = 'y1'*, *y2_key: str = 'y2'*, *is_query_key: str = 'is_query'*, *is_gallery_key: str = 'is_gallery'*)

Bases: *ImageLabeledDataset*, *IQueryGalleryLabeledDataset*

The annotated dataset of images having *query*/*gallery* split.

Note, that some datasets used as benchmarks in Metric Learning explicitly provide the splitting information (for example, `DeepFashion InShop` dataset), but some of them don't (for example, `CARS196` or `CUB200`). The validation idea for the latter is to perform *1 vs rest* validation, where every query is evaluated versus the whole validation dataset (except for this exact query).

So, if you want an item participate in validation as both: query and gallery, you should mark this item as `is_query == True` and `is_gallery == True`, as it's done in the *CARS196* or *CUB200* dataset.

__init__(*df: DataFrame*, *extra_data: Optional[Dict[str, Any]] = None*, *dataset_root: Optional[Union[Path, str]] = None*, *transform: Optional[Compose] = None*, *f_imread: Optional[Callable[[Union[Path, str, bytes]], Union[Image, ndarray]]] = None*, *cache_size: Optional[int] = 0*, *input_tensors_key: str = 'input_tensors'*, *labels_key: str = 'labels'*, *paths_key: str = 'paths'*, *categories_key: Optional[str] = 'categories'*, *sequence_key: Optional[str] = 'sequence'*, *x1_key: str = 'x1'*, *x2_key: str = 'x2'*, *y1_key: str = 'y1'*, *y2_key: str = 'y2'*, *is_query_key: str = 'is_query'*, *is_gallery_key: str = 'is_gallery'*)

Parameters

- **paths** – Paths to images. Will be concatenated with `dataset_root` if provided.
- **dataset_root** – Path to the images' dir, set `None` if you provided the absolute paths in your dataframe
- **bboxes** – Bounding boxes of images. Some of the images may not have bounding bboxes.
- **extra_data** – Dictionary containing records of some additional information.
- **transform** – Augmentations for the images, set `None` to perform only normalisation and casting to tensor
- **f_imread** – Function to read the images, pass `None` to pick it automatically based on provided transforms
- **cache_size** – Size of the dataset's cache
- **input_tensors_key** – Key to put tensors into the batches
- **index_key** – Key to put samples' ids into the batches
- **paths_key** – Key put paths into the batches # todo 522: remove

- **x1_key** – Key to put `x1` into the batches # todo 522: remove
- **x2_key** – Key to put `x2` into the batches # todo 522: remove
- **y1_key** – Key to put `y1` into the batches # todo 522: remove
- **y2_key** – Key to put `y2` into the batches # todo 522: remove

__getitem__(*idx: int*) → Dict[str, Any]

Parameters
item – Idx of the sample

Return type

Dictionary including the following keys

`self.labels_key`

get_query_ids() → LongTensor

get_gallery_ids() → LongTensor

get_labels() → ndarray

visualize(*item: int*, *color: Tuple[int, int, int] = (0, 0, 0)*) → ndarray

14.4 ImageQueryGalleryDataset

class oml.datasets.images.**ImageQueryGalleryDataset**(*df: DataFrame*, *extra_data: Optional[Dict[str, Any]] = None*, *dataset_root: Optional[Union[Path, str]] = None*, *transform: Optional[Compose] = None*, *f_imread: Optional[Callable[[Union[Path, str, bytes]], Union[Image, ndarray]]] = None*, *cache_size: Optional[int] = 0*, *input_tensors_key: str = 'input_tensors'*, *paths_key: str = 'paths'*, *categories_key: Optional[str] = 'categories'*, *sequence_key: Optional[str] = 'sequence'*, *x1_key: str = 'x1'*, *x2_key: str = 'x2'*, *y1_key: str = 'y1'*, *y2_key: str = 'y2'*, *is_query_key: str = 'is_query'*, *is_gallery_key: str = 'is_gallery'*)

Bases: *IVisualizableDataset*, *IQueryGalleryDataset*

The NOT annotated dataset of images having *query*/*gallery* split.

__init__(*df: DataFrame*, *extra_data: Optional[Dict[str, Any]] = None*, *dataset_root: Optional[Union[Path, str]] = None*, *transform: Optional[Compose] = None*, *f_imread: Optional[Callable[[Union[Path, str, bytes]], Union[Image, ndarray]]] = None*, *cache_size: Optional[int] = 0*, *input_tensors_key: str = 'input_tensors'*, *paths_key: str = 'paths'*, *categories_key: Optional[str] = 'categories'*, *sequence_key: Optional[str] = 'sequence'*, *x1_key: str = 'x1'*, *x2_key: str = 'x2'*, *y1_key: str = 'y1'*, *y2_key: str = 'y2'*, *is_query_key: str = 'is_query'*, *is_gallery_key: str = 'is_gallery'*)

__getitem__(*item: int*) → Dict[str, Any]

Parameters
item – Idx of the sample

Returns
`self.input_tensors_key self.index_key: int = item`

Return type
Dictionary including the following keys

get_query_ids() → LongTensor

get_gallery_ids() → LongTensor

visualize(*item: int*, *color: Tuple[int, int, int] = (0, 0, 0)*) → ndarray

14.5 EmbeddingPairsDataset

class oml.datasets.pairs.**EmbeddingPairsDataset**(*embeddings1: Tensor*, *embeddings2: Tensor*, *pair_1st_key: str = 'input_tensors_1'*, *pair_2nd_key: str = 'input_tensors_2'*, *index_key: str = 'idx'*)

Bases: *IPairsDataset*

Dataset to iterate over pairs of embeddings.

__init__(*embeddings1: Tensor*, *embeddings2: Tensor*, *pair_1st_key: str = 'input_tensors_1'*, *pair_2nd_key: str = 'input_tensors_2'*, *index_key: str = 'idx'*)

Parameters

- **embeddings1** – The first input embeddings
- **embeddings2** – The second input embeddings
- **pair_1st_key** – Key to put `embeddings1` into the batches
- **pair_2nd_key** – Key to put `embeddings2` into the batches
- **index_key** – Key to put samples' ids into the batches

__getitem__(*idx: int*) → Dict[str, Tensor]

Parameters
item – Idx of the sample

Return type

Dictionary with the following keys

`self.pairs_1st_key` `self.pairs_2nd_key` `self.index_key`

14.6 ImagePairsDataset

class oml.datasets.pairs.**ImagePairsDataset**(*paths1: ~typing.List[~pathlib.Path], paths2: ~typing.List[~pathlib.Path], bboxes1: ~typing.Optional[~typing.Sequence[~typing.Optional[~typing.Tuple[int, int, int, int]]]] = None, bboxes2: ~typing.Optional[~typing.Sequence[~typing.Optional[~typing.Tuple[int, int, int, int]]]] = None, transform: ~typing.Optional[~typing.Union[~albumentations.core.composition.Compose, ~torchvision.transforms.transforms.Compose]] = None, f_imread: ~typing.Callable[[~typing.Union[~pathlib.Path, str, bytes]], ~typing.Union[~PIL.Image.Image, ~numpy.ndarray]] = <function imread_pillow>, pair_1st_key: str = 'input_tensors_1', pair_2nd_key: str = 'input_tensors_2', index_key: str = 'idx', cache_size: ~typing.Optional[int] = 0*)

Bases: *IPairsDataset*

Dataset to iterate over pairs of images.

__init__(*paths1: ~typing.List[~pathlib.Path], paths2: ~typing.List[~pathlib.Path], bboxes1: ~typing.Optional[~typing.Sequence[~typing.Optional[~typing.Tuple[int, int, int, int]]]] = None, bboxes2: ~typing.Optional[~typing.Sequence[~typing.Optional[~typing.Tuple[int, int, int, int]]]] = None, transform: ~typing.Optional[~typing.Union[~albumentations.core.composition.Compose, ~torchvision.transforms.transforms.Compose]] = None, f_imread: ~typing.Callable[[~typing.Union[~pathlib.Path, str, bytes]], ~typing.Union[~PIL.Image.Image, ~numpy.ndarray]] = <function imread_pillow>, pair_1st_key: str = 'input_tensors_1', pair_2nd_key: str = 'input_tensors_2', index_key: str = 'idx', cache_size: ~typing.Optional[int] = 0*)

Parameters

- **paths1** – Paths to the 1st input images
- **paths2** – Paths to the 2nd input images
- **bboxes1** – Should be either `None` or a sequence of bboxes. If an image has `N` boxes, duplicate its path `N` times and provide bounding box for each of them. If you want to get an embedding for the whole image, set bbox to `None` for this particular image path. The format is `x1, y1, x2, y2`.
- **bboxes2** – The same as `bboxes2`, but for the second inputs.
- **transform** – Augmentations for the images, set `None` to perform only normalisation and casting to tensor
- **f_imread** – Function to read the images
- **pair_1st_key** – Key to put the 1st images into the batches
- **pair_2nd_key** – Key to put the 2nd images into the batches
- **index_key** – Key to put samples' ids into the batches
- **cache_size** – Size of the dataset's cache

__getitem__(*idx: int*) → Dict[str, Union[int, Dict[str, Any]]]

Parameters

item – Idx of the sample

Return type

Dictionary with the following keys

self.pairs_1st_key self.pairs_2nd_key self.index_key

CHAPTER

FIFTEEN

SAMPLERS

15.1 BalanceSampler

class oml.samplers.balance.**BalanceSampler**(*labels: Union[List[int], ndarray]*, *n_labels: int*, *n_instances: int*)

Bases: *IBatchSampler*

This sampler takes `n_instances` for each of the `n_labels` to form the batches. Thus, the batch size is `n_instances x n_labels`. This type of sampling can be found in the classical Person Re-Id paper - In Defense of the Triplet Loss for Person Re-Identification.

The strategy for the dataset with `L` unique labels is the following:

- Select `n_labels` of `L` labels for the 1st batch
- Select `n_instances` for each label for the 1st batch
- Select `n_labels` of `L - n_labels` remaining labels for 2nd batch
- Select `n_instances` instances for each label for the 2nd batch
- ...
- The epoch ends after `L // n_labels`.

Thus, in each epoch, all the labels will be selected once, but this does not mean that all the instances will be picked.

Behavior in corner cases:

- If some label does not contain `n_instances`, a choice will be made with repetition.
- If `L % n_labels != 0` then we drop the last batch.

__init__(*labels: Union[List[int], ndarray]*, *n_labels: int*, *n_instances: int*)

Parameters

- **labels** – List of the labels for each element in the dataset
- **n_labels** – The desired number of labels in a batch, should be > 1

- **n_instances** – The desired number of instances of each label in a batch, should be > 1

15.2 CategoryBalanceSampler

class oml.samplers.category_balance.**CategoryBalanceSampler**(*labels: Union[List[int], ndarray]*, *label2category: Dict[int, Union[str, int]]*, *n_categories: int*, *n_labels: int*, *n_instances: int*, *resample_labels: bool = False*, *weight_categories: bool = True*)

Bases: *IBatchSampler*

This sampler takes `n_instances` for each of the `n_labels` for each of the `n_categories` to form the batches. Thus, the batch size is `n_instances x n_labels x n_categories`.

Note, to form an epoch of batches we simply sample `L / n_labels` batches with repetition.

__init__(*labels: Union[List[int], ndarray]*, *label2category: Dict[int, Union[str, int]]*, *n_categories: int*, *n_labels: int*, *n_instances: int*, *resample_labels: bool = False*, *weight_categories: bool = True*)

Parameters

- **labels** – Labels to sample from
- **label2category** – Mapping from label to category
- **n_categories** – The desired number of categories to sample for each batch
- **n_labels** – The desired number of labels to sample for each category in batch
- **n_instances** – The desired number of samples to sample for each label in batch
- **resample_labels** – If `True` sample with repetition otherwise, otherwise raise an error in case of the labels lack in any category
- **weight_categories** – If `True` sample categories for each batch with weights proportional to the number of unique labels in the categories

15.3 DistinctCategoryBalanceSampler

class oml.samplers.distinct_category_balance.**DistinctCategoryBalanceSampler**(*labels: Union[List[int], ndarray]*, *label2category: Dict[int, Union[str, int]]*, *n_categories: int*, *n_labels: int*, *n_instances: int*, *epoch_size: int*)

Bases: *IBatchSampler*

This sampler takes `n_instances` for each of the `n_labels` for each of the `n_categories` to form the batches. Thus, the batch size is `n_instances x n_labels x n_categories`.

The strategy for the dataset with `L` unique labels and `C` unique categories is the following:

- Select `n_categories` of `C` for the 1st batch
- Select `n_labels` for each of the chosen categories for the 1st batch
- Select `n_instances` for each of the chosen labels for the 1st batch
- Define the set of available for the 2nd batch labels `L^`: these are all the labels `L` except the ones chosen for the 1st batch
- Define set of available categories `C^`: these are all the categories corresponding to labels from `L^`
- Select `n_categories` from `C^` for the 2nd batch
- Select `n_labels` for each category from `L^` for the 2nd batch
- Select `n_instances` for each label for the 2nd batch
- ...
- Epoch ends after `epoch_size` steps

Behavior in corner cases:

- If all the categories were chosen before `epoch_size` steps, the sampler resets its state and goes on sampling

  from the first step.

- If some class does not contain `n_instances`, a choice will be made with repetition.
- If the chosen category does not contain unused `n_labels`, all the unused labels will be added to a batch and

  the missing ones will be sampled from the used labels without repetition.

- If `L % n_labels == 1` then one of the labels must be dropped because we always want to have more than 1 label

  in a batch to be able to form positive pairs later on.

__init__(*labels: Union[List[int], ndarray]*, *label2category: Dict[int, Union[str, int]]*, *n_categories: int*, *n_labels: int*, *n_instances: int*, *epoch_size: int*)

Parameters

- **labels** – Labels to sample from
- **label2category** – Mapping from label to category
- **n_categories** – The desired number of categories to sample for each batch
- **n_labels** – The desired number of labels to sample for each category in batch
- **n_instances** – The desired number of samples to sample for each label in batch
- **epoch_size** – The desired number of batches in epoch

CHAPTER

SIXTEEN

MINERS

16.1 AllTripletsMiner

class oml.miners.inbatch_all_tri.**AllTripletsMiner**(*max_output_triplets: int = 9223372036854775807*, *device: str = 'cpu'*)

Bases: *ITripletsMinerInBatch*

This miner selects all the possible triplets for the given batch.

__init__(*max_output_triplets: int = 9223372036854775807*, *device: str = 'cpu'*)

Parameters

- **max_output_triplets** – Number of all of the possible triplets in the batch can be very large, so we can limit them vis this parameter.
- **device** – the device where to perform computations.

sample(*features: Tensor*, *labels: Union[List[int], Tensor]*) → Tuple[Tensor, Tensor, Tensor]

Parameters

- **features** – Features with the shape of `[batch_size, feature_size]`
- **labels** – Labels with the size of `batch_size`

Returns
Batch of triplets

16.2 HardTripletsMiner

class oml.miners.inbatch_hard_tri.**HardTripletsMiner**

Bases: *ITripletsMinerInBatch*

This miner selects the hardest triplets based on the distances between the features:

- The hardest *positive* sample has the *maximal* distance to the anchor sample
- The hardest *negative* sample has the *minimal* distance to the anchor sample

__init__()

sample(*features: Tensor*, *labels: Union[List[int], Tensor]*) → Tuple[Tensor, Tensor, Tensor]

Parameters

- **features** – Features with the shape of `[batch_size, feature_size]`
- **labels** – Labels with the size of `batch_size`

Returns

Batch of triplets

16.3 TripletMinerWithMemory

class oml.miners.cross_batch.**TripletMinerWithMemory**(*bank_size_in_batches: int*, *tri_expand_k: int*)

Bases: *ITripletsMiner*

This miner has a memory bank that allows to sample not only the triplets from the original batch, but also add batches obtained from both the bank and the original batch.

__init__(*bank_size_in_batches: int*, *tri_expand_k: int*)

Parameters

- **bank_size_in_batches** – The size of the bank calculated in the number batches
- **tri_expand_k** – This parameter defines how many triplets we sample from the bank. Specifically, we return `tri_expand_k * number of original triplets`. In particular, if `tri_expand_k == 1` we sample no triplets from the bank

sample(*features: Tensor*, *labels: Tensor*) → Tuple[Tensor, Tensor, Tensor, Tensor]

Parameters

- **features** – Features with the shape of `(batch_size, feat_dim)`
- **labels** – Labels with the size of `batch_size`

Returns

Triplets made from the original batch and those that were combined from the bank and the batch. We also return an indicator of whether triplet was obtained from the original batch. So, output is the following `(anchor, positive, negative, indicators)`

16.4 HardClusterMiner

class oml.miners.inbatch_hard_cluster.**HardClusterMiner**

Bases: *ITripletsMiner*

This miner selects the hardest triplets based on the distance to mean vectors: anchor is a mean vector of features of i-th label in the batch, the hardest positive sample is the most distant from the anchor sample of anchor's label, the hardest negative sample is the closest mean vector of other labels.

The batch must contain `n_instances` for `n_labels` where both values higher than 1.

__init__()

sample(*features: Tensor*, *labels: Union[List[int], Tensor]*) → Tuple[Tensor, Tensor, Tensor]

This method samples the hardest triplets in the batch.

Parameters

- **features** – Tensor with the shape of `[batch_size, embed_dim]` that contains `n_instances` for each of `n_labels`
- **labels** – Labels with the size of `batch_size`

Returns

`n_labels` triplets in the form of `(mean_vector, positive, negative_mean_vector)`

16.5 NHardTripletsMiner

class oml.miners.inbatch_nhard_tri.**NHardTripletsMiner**(*n_positive: Union[Tuple[int, int], List[int], int] = 1*, *n_negative: Union[Tuple[int, int], List[int], int] = 1*)

Bases: *ITripletsMinerInBatch*

This miner selects hard triplets based on distances between features:

- hard *positive* samples have large distance to the anchor sample
- hard *negative* samples have small distance to the anchor sample

Toward the end of the training, annotation errors can affect final metric. If you are not sure about the quality of your dataset, you can use range instead of integer value for parameters and exclude combinations with the largest distances. For example instead picking 5 positive examples, you can use examples from the 2nd hardest to the 5th one.

__init__(*n_positive: Union[Tuple[int, int], List[int], int] = 1*, *n_negative: Union[Tuple[int, int], List[int], int] = 1*)

Parameters

- **n_positive** – keep `n_positive` positive samples with large distances. If the value is a range, minimal value has to be less than the available amount of labels in batches
- **n_negative** – keep `n_negative` negative pipelines with small distances

Note: If both parameters are 1, the miner is equivalent to `HardTripletsMiner`. If both parameters are large enough, the miner can be equivalent to `AllTripletsMiner`

sample(*features: Tensor*, *labels: Union[List[int], Tensor]*) → Tuple[Tensor, Tensor, Tensor]

Parameters

- **features** – Features with the shape `[batch_size, feature_size]`
- **labels** – Labels with the size of `batch_size`

Returns
Batch of triplets

16.6 MinerWithBank

class oml.miners.miner_with_bank.**MinerWithBank**(*bank_size_in_batches: int*, *miner:* NHardTripletsMiner, *need_logs: bool = True*)

Bases: *ITripletsMiner*

This is a class for cross-batch memory. This implementation uses only samples from the current batch as anchors and finds positive and negative pairs from the bank and the current batch using miner.

__init__(*bank_size_in_batches: int*, *miner:* NHardTripletsMiner, *need_logs: bool = True*)

Parameters

- **bank_size_in_batches** – Size of the bank.
- **miner** – Miner, for now we only support `NHardTripletsMiner`
- **need_logs** – Set `True` if you want to track logs.

sample(*features: Tensor*, *labels: Tensor*) → Tuple[Tensor, Tensor, Tensor]

Parameters

- **features** – Features with the shape `[batch_size, features_dim]`
- **labels** – Labels with the size of `batch_size`

Returns
anchor, positive, negative

Return type
Batch of triplets in the following order

CHAPTER

SEVENTEEN

LOSSES

17.1 TripletLoss

class oml.losses.triplet.**TripletLoss**(*margin: Optional[float]*, *reduction: str = 'mean'*, *need_logs: bool = False*)

Bases: `Module`

Class, which combines classical *TripletMarginLoss* and *SoftTripletLoss*. The idea of *SoftTripletLoss* is the following: instead of using the classical formula `loss = relu(margin + positive_distance - negative_distance)` we use `loss = log1p(exp(positive_distance - negative_distance))`. It may help to solve the often problem when *TripletMarginLoss* converges to it's margin value (also known as *dimension collapse*).

__init__(*margin: Optional[float]*, *reduction: str = 'mean'*, *need_logs: bool = False*)

Parameters

- **margin** – Margin value, set `None` to use *SoftTripletLoss*
- **reduction** – `mean`, `sum` or `none`
- **need_logs** – Set `True` if you want to store logs

forward(*anchor: Tensor*, *positive: Tensor*, *negative: Tensor*) → Tensor

Parameters

- **anchor** – Anchor features with the shape of `(batch_size, feat)`
- **positive** – Positive features with the shape of `(batch_size, feat)`
- **negative** – Negative features with the shape of `(batch_size, feat)`

Returns
Loss value

17.2 TripletLossPlain

class oml.losses.triplet.**TripletLossPlain**(*margin: Optional[float]*, *reduction: str = 'mean'*, *need_logs: bool = False*)

Bases: `Module`

The same as *TripletLoss*, but works with anchor, positive and negative features stacked together.

__init__(*margin: Optional[float]*, *reduction: str = 'mean'*, *need_logs: bool = False*)

Parameters

- **margin** – Margin value, set `None` to use *SoftTripletLoss*
- **reduction** – `mean`, `sum` or `none`
- **need_logs** – Set `True` if you want to store logs

forward(*features: Tensor*) → Tensor

Parameters
features – Features with the shape of `[batch_size, feat]` with the following structure: *0,1,2* are indices of the 1st triplet, *3,4,5* are indices of the 2nd triplet, and so on. Thus, the features contains `(N / 3)` triplets

Returns
Loss value

17.3 TripletLossWithMiner

class oml.losses.triplet.**TripletLossWithMiner**(*margin: ~typing.Optional[float]*, *miner: ~oml.interfaces.miners.ITripletsMiner = <oml.miners.inbatch_all_tri.AllTripletsMiner object>*, *reduction: str = 'mean'*, *need_logs: bool = False*)

Bases: *ITripletLossWithMiner*

This class combines *Miner* and *TripletLoss*.

__init__(*margin: ~typing.Optional[float]*, *miner: ~oml.interfaces.miners.ITripletsMiner = <oml.miners.inbatch_all_tri.AllTripletsMiner object>*, *reduction: str = 'mean'*, *need_logs: bool = False*)

Parameters

- **margin** – Margin value, set `None` to use *SoftTripletLoss*
- **miner** – A miner that implements the logic of picking triplets to pass them to the triplet loss.
- **reduction** – `mean`, `sum` or `none`
- **need_logs** – Set `True` if you want to store logs

forward(*features: Tensor*, *labels: Union[Tensor, List[int]]*) → Tensor

Parameters

- **features** – Features with the shape `[batch_size, feat]`
- **labels** – Labels with the size of `batch_size`

Returns
Loss value

17.4 SurrogatePrecision

class oml.losses.surrogate_precision.**SurrogatePrecision**(*k: int*, *temperature1: float = 1.0*, *temperature2: float = 0.01*, *reduction: str = 'mean'*)

Bases: `Module`

This loss is a differentiable approximation of Precision@k metric.

The loss is described in the following paper under a bit different name: Recall@k Surrogate Loss with Large Batches and Similarity Mixup.

The idea is that we express the formula for Precision@k using two step functions (aka Heaviside functions). Then we approximate them using two sigmoid functions with temperatures. The smaller temperature the close sigmoid to the step function, but the gradients are sparser, and vice versa. In the original paper *t1 = 1.0* and *t2 = 0.01* have been used.

__init__(*k: int*, *temperature1: float = 1.0*, *temperature2: float = 0.01*, *reduction: str = 'mean'*)

Parameters

- **k** – Parameter of Precision@k.
- **temperature1** – Scaling factor for the 1st sigmoid, see docs above.
- **temperature2** – Scaling factor for the 2nd sigmoid, see docs above.
- **reduction** – `mean`, `sum` or `none`

forward(*features: Tensor*, *labels: Tensor*) → Tensor

Parameters

- **features** – Features with the shape of `[batch_size, feature_size]`
- **labels** – Labels with the size of `batch_size`

Returns
Loss value

17.5 ArcFaceLoss

class oml.losses.arcface.**ArcFaceLoss**(*in_features: int*, *num_classes: int*, *m: float = 0.5*, *s: float = 64*, *smoothing_epsilon: float = 0*, *label2category: Optional[Dict[Any, Any]] = None*, *reduction: str = 'mean'*)

Bases: `Module`

ArcFace loss from paper with possibility to use label smoothing. It contains projection size of `num_features x num_classes` inside itself. Please make sure that class labels started with 0 and ended as `num_classes` - 1.

__init__(*in_features: int*, *num_classes: int*, *m: float = 0.5*, *s: float = 64*, *smoothing_epsilon: float = 0*, *label2category: Optional[Dict[Any, Any]] = None*, *reduction: str = 'mean'*)

Parameters

- **in_features** – Input feature size
- **num_classes** – Number of classes in train set
- **m** – Margin parameter for ArcFace loss. Usually you should use 0.3-0.5 values for it
- **s** – Scaling parameter for ArcFace loss. Usually you should use 30-64 values for it
- **smoothing_epsilon** – Label smoothing effect strength
- **label2category** – Optional, mapping from label to its category. If provided, label smoothing will redistribute `smoothing_epsilon` only inside the category corresponding to the sample's ground truth label
- **reduction** – CrossEntropyLoss reduction

17.6 ArcFaceLossWithMLP

class oml.losses.arcface.**ArcFaceLossWithMLP**(*in_features: int*, *num_classes: int*, *mlp_features: List[int]*, *m: float = 0.5*, *s: float = 64*, *smoothing_epsilon: float = 0*, *label2category: Optional[Dict[Any, Any]] = None*, *reduction: str = 'mean'*)

Bases: `Module`

Almost the same as `ArcFaceLoss`, but also has MLP projector before the loss. You may want to use `ArcFaceLossWithMLP` to boost the expressive power of ArcFace loss during the training (for example, in a multi-head setup it may be a good idea to have task-specific projectors in each of the losses). Note, the criterion does not exist during the validation time. Thus, if you want to keep your MLP layers, you should create them as a part of the model you train.

__init__(*in_features: int*, *num_classes: int*, *mlp_features: List[int]*, *m: float = 0.5*, *s: float = 64*, *smoothing_epsilon: float = 0*, *label2category: Optional[Dict[Any, Any]] = None*, *reduction: str = 'mean'*)

Parameters

- **in_features** – Input feature size
- **num_classes** – Number of classes in train set
- **mlp_features** – Layers sizes for MLP before ArcFace
- **m** – Margin parameter for ArcFace loss. Usually you should use 0.3-0.5 values for it

- **s** – Scaling parameter for ArcFace loss. Usually you should use 30-64 values for it
- **smoothing_epsilon** – Label smoothing effect strength
- **label2category** – Optional, mapping from label to its category. If provided, label smoothing will redistribute `smoothing_epsilon` only inside the category corresponding to the sample's ground truth label
- **reduction** – CrossEntropyLoss reduction

17.7 label_smoothing

oml.functional.label_smoothing.**label_smoothing**(*y: Tensor*, *num_classes: int*, *epsilon: float = 0.2*, *categories: Optional[Tensor] = None*) → Tensor

This function is doing label smoothing. You can also use modified version, where the label is smoothed only for the category corresponding to sample's ground truth label. To use this, you should provide the `categories` argument: vector, for which i-th entry is a corresponding category for label `i`.

Parameters

- **y** – Ground truth labels with the size of batch_size where each element is from 0 (inclusive) to num_classes (exclusive).
- **num_classes** – Number of classes in total
- **epsilon** – Power of smoothing. The biggest value in OHE-vector will be `1 - epsilon + 1 / num_classes` after the transformation
- **categories** – Vector for which i-th entry is a corresponding category for label `i`. Optional, used for category-based label smoothing. In that case the biggest value in OHE-vector will be `1 - epsilon + 1 / num_classes_of_the_same_category`, labels outside of the category will not change

CHAPTER

EIGHTEEN

MODELS

18.1 ViTExtractor

class oml.models.vit_dino.extractor.**ViTExtractor**(*weights: Optional[Union[Path, str]]*, *arch: str*, *normalise_features: bool*, *use_multi_scale: bool = False*)

Bases: *IExtractor*

The base class for the extractors that follow VisualTransformer architecture.

__init__(*weights: Optional[Union[Path, str]]*, *arch: str*, *normalise_features: bool*, *use_multi_scale: bool = False*)

Parameters

- **weights** – Path to weights or a special key to download pretrained checkpoint, use `None` to randomly initialize model's weights. You can check the available pretrained checkpoints in `self.pretrained_models`.
- **arch** – Might be one of `vits8`, `vits16`, `vitb8`, `vitb16`. You can check all the available options in `self.constructors`
- **normalise_features** – Set `True` to normalise output features
- **use_multi_scale** – Set `True` to use multiscale (the analogue of test time augmentations)

draw_attention(*image: Union[Image, ndarray]*) → ndarray

Parameters

image – An image with pixel values in the range of `[0..255]`.

Returns
An image with drawn attention maps.

Visualization of the multi-head attention on a particular image.

property feat_dim: int

The only method that obligatory to implemented.

18.2 ViTCLIPExtractor

class oml.models.vit_clip.extractor.**ViTCLIPExtractor**(*weights: Optional[str]*, *arch: str*, *normalise_features: bool = True*)

Bases: *IExtractor*

__init__(*weights: Optional[str]*, *arch: str*, *normalise_features: bool = True*)

Parameters

- **weights** – Path to weights or special key for pretrained ones or `None` for random initialization. You can check available pretrained checkpoints in `ViTCLIPExtractor.pretrained_models`.
- **arch** – Might be one of `vitb16_224`, `vitb32_224`, `vitl14_224`, `vitl14_336`.
- **normalise_features** – Set `True` to normalise output features

property feat_dim: int

The only method that obligatory to implemented.

18.3 ResnetExtractor

class oml.models.resnet.extractor.**ResnetExtractor**(*weights: Optional[Union[Path, str]]*, *arch: str*, *gem_p: Optional[float]*, *remove_fc: bool*, *normalise_features: bool*)

Bases: *IExtractor*

The base class for the extractors that follow ResNet architecture.

__init__(*weights: Optional[Union[Path, str]]*, *arch: str*, *gem_p: Optional[float]*, *remove_fc: bool*, *normalise_features: bool*)

Parameters

- **weights** – Path to weights or a special key to download pretrained checkpoint, use `None` to randomly initialize model's weights. You can check the available pretrained checkpoints in `self.pretrained_models`.
- **arch** – Different types of ResNet, please, check `self.constructors`
- **gem_p** – Value of power in *Generalized Mean Pooling* that we use as the replacement for the default one (if `gem_p == 1` or `None` it's just a normal average pooling and if `gem_p -> inf` it's max-pooling)
- **remove_fc** – Set `True` if you want to remove the last fully connected layer. Note, that having this layer is obligatory for calling `draw_gradcam()` method
- **normalise_features** – Set `True` to normalise output features

draw_gradcam(*image: Union[ndarray, Image]*) → Union[ndarray, Image]

Parameters
image – An image with pixel values in the range of `[0..255]`.

Returns
An image with drawn gradients.

Visualization of the gradients on a particular image using GradCam.

property feat_dim: `int`

The only method that obligatory to implemented.

18.4 ExtractorWithMLP

class oml.models.meta.projection.**ExtractorWithMLP**(*extractor: IExtractor, mlp_features: List[int], weights: Optional[Union[Path, str]] = None, train_backbone: bool = False*)

Bases: *IExtractor*, *IFreezable*

Class-wrapper for extractors which an additional MLP.

__init__(*extractor: IExtractor, mlp_features: List[int], weights: Optional[Union[Path, str]] = None, train_backbone: bool = False*)

Parameters

- **extractor** – Instance of `IExtractor` (e.g. `ViTExtractor`)
- **mlp_features** – Sizes of projection layers
- **weights** – Path to weights file or `None` for random initialization
- **train_backbone** – set `False` if you want to train only MLP head

18.5 LinearTrivialDistanceSiamese

class oml.models.meta.siamese.**LinearTrivialDistanceSiamese**(*feat_dim: int, identity_init: bool*)

Bases: *IPairwiseModel*

This model is a useful tool mostly for development.

__init__(*feat_dim: int, identity_init: bool*)

Parameters

- **feat_dim** – Expected size of each input.
- **identity_init** – If `True`, models' weights initialised in a way when the model simply estimates L2 distance between the original embeddings.

forward(*x1: Tensor, x2: Tensor*) → Tensor

Parameters

- **x1** – Embedding with the shape of `[batch_size, feat_dim]`
- **x2** – Embedding with the shape of `[batch_size, feat_dim]`

Returns
Distance between transformed inputs.

18.6 TrivialDistanceSiamese

class oml.models.meta.siamese.**TrivialDistanceSiamese**(*extractor:* IExtractor)

Bases: *IPairwiseModel*

This model is a useful tool mostly for development.

__init__(*extractor:* IExtractor) → None

Parameters
extractor – Instance of `IExtractor` (e.g. `ViTExtractor`)

forward(*x1: Tensor*, *x2: Tensor*) → Tensor

Parameters

- **x1** – The first input.
- **x2** – The second input.

Returns
Distance between inputs.

18.7 ConcatSiamese

class oml.models.meta.siamese.**ConcatSiamese**(*extractor:* IExtractor, *mlp_hidden_dims: List[int]*, *use_tta: bool = False*, *weights: Optional[Union[Path, str]] = None*)

Bases: *IPairwiseModel*, *IFreezable*

This model concatenates two inputs and passes them through a given backbone and applyies a head after that.

__init__(*extractor:* IExtractor, *mlp_hidden_dims: List[int]*, *use_tta: bool = False*, *weights: Optional[Union[Path, str]] = None*) → None

Parameters

- **extractor** – Instance of `IExtractor` (e.g. `ViTExtractor`)
- **mlp_hidden_dims** – Hidden dimensions of the head
- **use_tta** – Set `True` if you want to average the results obtained by two different orders of concatenating input images. Affects only `self.predict()` method.
- **weights** – Path to weights file or `None` for random initialization

forward(*x1: Tensor*, *x2: Tensor*) → Tensor

Parameters

- **x1** – The first input.
- **x2** – The second input.

predict(*x1: Tensor*, *x2: Tensor*) → Tensor

While `self.forward()` is called during training, this method is called during inference or validation time. For example, it allows application of some activation, which was a part of a loss function during the training.

Parameters

- **x1** – The first input.
- **x2** – The second input.

freeze() → None

Function for freezing. You can use it to partially freeze a model.

unfreeze() → None

Function for unfreezing. You can use it to unfreeze a model.

CHAPTER

NINETEEN

METRICS

19.1 EmbeddingMetrics

class oml.metrics.embeddings.**EmbeddingMetrics**(*embeddings_key: str = 'embeddings', labels_key: str = 'labels', is_query_key: str = 'is_query', is_gallery_key: str = 'is_gallery', extra_keys: Tuple[str, ...] = (), cmc_top_k: Tuple[int, ...] = (5,), precision_top_k: Tuple[int, ...] = (5,), map_top_k: Tuple[int, ...] = (5,), fmr_vals: Tuple[float, ...] = (), pcf_variance: Tuple[float, ...] = (0.5,), categories_key: Optional[str] = None, sequence_key: Optional[str] = None, postprocessor: Optional[IDistancesPostprocessor] = None, metrics_to_exclude_from_visualization: Iterable[str] = (), return_only_overall_category: bool = False, visualize_only_overall_category: bool = True, verbose: bool = True*)

Bases: `IMetricVisualisable`

This class accumulates the information from the batches and embeddings produced by the model at every batch in epoch. After all the samples have been stored, you can call the function which computes retrievals metrics. To get the needed information from the batches, it uses keys which have to be provided as init arguments. Please, check the usage example in *Readme*.

__init__(*embeddings_key: str = 'embeddings', labels_key: str = 'labels', is_query_key: str = 'is_query', is_gallery_key: str = 'is_gallery', extra_keys: Tuple[str, ...] = (), cmc_top_k: Tuple[int, ...] = (5,), precision_top_k: Tuple[int, ...] = (5,), map_top_k: Tuple[int, ...] = (5,), fmr_vals: Tuple[float, ...] = (), pcf_variance: Tuple[float, ...] = (0.5,), categories_key: Optional[str] = None, sequence_key: Optional[str] = None, postprocessor: Optional[IDistancesPostprocessor] = None, metrics_to_exclude_from_visualization: Iterable[str] = (), return_only_overall_category: bool = False, visualize_only_overall_category: bool = True, verbose: bool = True*)

Parameters

- **embeddings_key** – Key to take the embeddings from the batches
- **labels_key** – Key to take the labels from the batches
- **is_query_key** – Key to take the information whether every batch sample belongs to the query
- **is_gallery_key** – Key to take the information whether every batch sample belongs to the gallery
- **extra_keys** – Keys to accumulate some additional information from the batches
- **cmc_top_k** – Values of `k` to calculate `cmc@k` (*Cumulative Matching Characteristic*)
- **precision_top_k** – Values of `k` to calculate `precision@k`
- **map_top_k** – Values of `k` to calculate `map@k` (*Mean Average Precision*)
- **fmr_vals** – Values of `fmr` (measured in quantiles) to calculate `fnmr@fmr` (*False Non Match Rate at the given False Match Rate*). For example, if `fmr_values` is (0.2, 0.4) we will calculate `fnmr@fmr=0.2` and `fnmr@fmr=0.4`. Note, computing this metric requires additional memory overhead, that is why it's turned off by default.
- **pcf_variance** – Values in range [0, 1]. Find the number of components such that the amount of variance that needs to be explained is greater than the percentage specified by `pcf_variance`.
- **categories_key** – Key to take the samples' categories from the batches (if you have ones)
- **sequence_key** – Key to take sequence ids from the batches (if you have ones)
- **postprocessor** – Postprocessor which applies some techniques like query reranking
- **metrics_to_exclude_from_visualization** – Names of the metrics to exclude from the visualization. It will not affect calculations.
- **return_only_overall_category** – Set `True` if you want to return only the aggregated metrics
- **visualize_only_overall_category** – Set `False` if you want to visualize each category separately
- **verbose** – Set `True` if you want to print metrics

setup(*num_samples: int*) → None

Method for preparing metrics to work: memory allocation, placeholder preparation, etc. Has to be called before the first call of `self.update_data()`.

update_data(*data_dict: Dict[str, Any], indices: Optional[List[int]] = None*) → None

Parameters

- **data_dict** – Batch of data containing records of the same size: `bs`.

- **indices** – Global indices of the elements in your records within the range of `(0, dataset_size - 1)`. Indices are needed in DDP (because data is gathered shuffled, additionally you may also get some duplicates due to padding). In the single device regime it's may be useful if you accumulate data in shuffled order.

compute_metrics() → Dict[Union[str, int], Dict[str, Dict[Union[int, float], Union[float, Tensor]]]]

The output must be in the following format:

```
{
    "self.overall_categories_key": {"metric1": ..., "metric2": ...},
    "category1": {"metric1": ..., "metric2": ...},
    "category2": {"metric1": ..., "metric2": ...}
}
```

Where `category1` and `category2` are optional.

get_plot_for_queries(*query_ids: List[int]*, *n_instances: int*, *verbose: bool = True*) → Figure

Visualize the predictions for the query with the indicies <query_ids>.

Parameters

- **query_ids** – Index of the query
- **n_instances** – Amount of the predictions to show
- **verbose** – wether to show image paths or not

get_worst_queries_ids(*metric_name: str*, *n_queries: int*) → List[int]

get_plot_for_worst_queries(*metric_name: str*, *n_queries: int*, *n_instances: int*, *verbose: bool = False*) → Figure

visualize() → Tuple[Collection[Figure], Collection[str]]

Visualize worst queries by all the available metrics.

19.2 calc_retrieval_metrics

oml.functional.metrics.**calc_retrieval_metrics**(*retrieved_ids: LongTensor*, *gt_ids: List[LongTensor]*, *cmc_top_k: Tuple[int, ...] = (5,)*, *precision_top_k: Tuple[int, ...] = (5,)*, *map_top_k: Tuple[int, ...] = (5,)*, *reduce: bool = True*) → Dict[str, Dict[Union[int, float], Union[float, Tensor]]]

Function to count different retrieval metrics.

Parameters

- **retrieved_ids** – Top N gallery ids retrieved for every query with the shape of `[n_query, top_n]`. Every element is within the range `(0, n_gallery - 1)`.
- **gt_ids** – Gallery ids relevant to every query, list of `n_query` elements where every element may have an arbitrary length. Every element is within the range `(0, n_gallery - 1)`
- **cmc_top_k** – Values of `k` to calculate `cmc@k` (*Cumulative Matching Characteristic*)
- **precision_top_k** – Values of `k` to calculate `precision@k`
- **map_top_k** – Values of `k` to calculate `map@k` (*Mean Average Precision*)
- **reduce** – If `False` return metrics for each query without averaging

Returns
Metrics dictionary.

19.3 calc_topological_metrics

oml.functional.metrics.**calc_topological_metrics**(*embeddings: Tensor*, *pcf_variance: Tuple[float, ...]*) → Dict[str, Dict[Union[int, float], Union[float, Tensor]]]

Function to evaluate different topological metrics.

Parameters

- **embeddings** – Embeddings matrix with the shape of `[n_embeddings, embeddings_dim]`.
- **pcf_variance** – Values in range [0, 1]. Find the number of components such that the amount of variance that needs to be explained is greater than the percentage specified by `pcf_variance`.

Returns
Metrics dictionary.

19.4 calc_cmc

oml.functional.metrics.**calc_cmc**(*gt_tops: Tensor*, *top_k: Tuple[int, ...]*) → List[Tensor]

Function to compute Cumulative Matching Characteristics (CMC) at cutoffs `top_k`.

`cmc@k` for a given query equals to 1 if there is at least 1 instance related to this query in top `k` gallery instances sorted by distances to the query, and 0 otherwise. The final `cmc@k` could be obtained by averaging the results calculated for each query.

Parameters

- **gt_tops** – Matrix where the `(i, j)` element indicates if `j`-th gallery sample is related to `i`-th query or not. Obtained from the full ground truth matrix by taking `max(top_k)` elements with the smallest distances to the corresponding queries.
- **top_k** – Values of `k` to calculate `cmc@k`.

Returns
List of `cmc@k` tensors.

$$\text{cmc@}k = \begin{cases} 1, & \text{if top-}k \text{ ranked gallery samples include an output relevant to the query,} \\ 0, & \text{otherwise.} \end{cases}$$

Example

```
>>> gt_tops = torch.tensor([
...                         [1, 0],
...                         [0, 1],
...                         [0, 0]
... ], dtype=torch.bool)
>>> calc_cmc(gt_tops, top_k=(1, 2))
[tensor([1., 0., 0.]), tensor([1., 1., 0.])]
```

19.5 calc_precision

oml.functional.metrics.**calc_precision**(*gt_tops: Tensor*, *n_gt: Tensor*, *top_k: Tuple[int, ...]*) → List[Tensor]

Function to compute Precision at cutoffs `top_k`.

`precision@k` for a given query is a fraction of the relevant gallery instances among the top `k` instances sorted by distances from the query to the gallery. The final `precision@k` could be obtained by averaging the results calculated for each query.

Parameters

- **gt_tops** – Matrix where the `(i, j)` element indicates if `j`-th gallery sample is related to `i`-th query or not. Obtained from the full ground truth matrix by taking `max(top_k)` elements with the smallest distances to the corresponding queries.
- **n_gt** – Array where the `i`-th element is the total number of elements in the gallery relevant to `i`-th query.
- **top_k** – Values of `k` to calculate `precision@k`.

Returns

List of `precision@k` tensors.

Given a list $g = [g_1, \ldots, g_k]$ of ground truth top k closest elements from the gallery to a given query (g_i is 1 if i-th element from the gallery is relevant to the query and 0 otherwise), and the total number of relevant elements from the gallery n, the precision@k for the query is defined as

$$\text{precision@}k = \frac{1}{\min(k, n)} \sum_{i=1}^{k} g_i$$

It's worth mentioning that OML version of precision@k differs from the commonly used by the denominator of the fraction. The OML version takes into account the total amount of relevant elements in the gallery, so it will not penalize the ideal model if $n < k$.

For instance, let $n = 3$ and $g = [1, 1, 1, 0, 0]$. Then by using the common definition of precision@k we get

$$\text{precision@}1 = \frac{1}{1}, \text{precision@}2 = \frac{2}{2}, \text{precision@}3 = \frac{3}{3}, \tag{19.1}$$

$$\text{precision@}4 = \frac{3}{4}, \text{precision@}5 = \frac{3}{5}, \text{precision@}6 = \frac{3}{6} \tag{19.2}$$

(19.3)

But with OML definition of precision@k we get

$$\text{precision@}1 = \frac{1}{1}, \text{precision@}2 = \frac{2}{2}, \text{precision@}3 = \frac{3}{3} \tag{19.4}$$

$$\text{precision@}4 = \frac{3}{3}, \text{precision@}5 = \frac{3}{3}, \text{precision@}6 = \frac{3}{3} \tag{19.5}$$

(19.6)

See:

Evaluation measures (information retrieval). Precision@k

Example

```
>>> gt_tops = torch.tensor([
...                         [1, 0],
...                         [0, 1],
...                         [0, 0]
... ], dtype=torch.bool)
>>> n_gt = torch.tensor([2, 3, 5])
>>> calc_precision(gt_tops, n_gt, top_k=(1, 2))
[tensor([1., 0., 0.]), tensor([0.5000, 0.5000, 0.0000])]
```

19.6 calc_map

oml.functional.metrics.**calc_map**(*gt_tops: Tensor*, *n_gt: Tensor*, *top_k: Tuple[int, ...]*) → List[Tensor]

Function to compute Mean Average Precision (MAP) at cutoffs `top_k`.

`map@k` for a given query is the average value of the `precision` considered as a function of the `recall`. The final `map@k` could be obtained by averaging the results calculated for each query.

Parameters

- **gt_tops** – Matrix where the `(i, j)` element indicates if `j`-th gallery sample is related to `i`-th query or not. Obtained from the full ground truth matrix by taking `max(top_k)` elements with the smallest distances to the corresponding queries.
- **n_gt** – Array where the `i`-th element is the total number of elements in the gallery relevant to `i`-th query.
- **top_k** – Values of `k` to calculate `map@k`.

Returns

List of `map@k` tensors.

Given a list $g = [g_1, \ldots, g_k]$ of ground truth top k closest elements from the gallery to a given query (g_i is 1 if i-th element from the gallery is relevant to the query and 0 otherwise), and the total number of relevant elements from the gallery n, the map@k for the query is defined as

$$\text{map@}k = \frac{1}{n_k} \sum_{i=1}^{k} \frac{n_i}{i} \times \text{rel}(i)$$

where $\text{rel}(i)$ is 1 if i-th element from the top i closest elements from the gallery to the query is relevant to the query, and 0 otherwise; and $n_i = \sum_{j=1}^{i} g_j$, which is the number of the relevant predictions among the first i outputs.

See:

Evaluation measures (information retrieval). Mean Average Precision

Mean Average Precision (MAP) For Recommender Systems

Example

```
>>> gt_tops = torch.tensor([
...                         [1, 0],
...                         [0, 1],
...                         [0, 0]
... ], dtype=torch.bool)
>>> n_gt = torch.tensor([2, 3, 5])
>>> calc_map(gt_tops, n_gt, top_k=(1, 2))
[tensor([1., 0., 0.]), tensor([1.0000, 0.5000, 0.0000])]
```

19.7 calc_fnmr_at_fmr

oml.functional.metrics.**calc_fnmr_at_fmr**(*pos_dist: Tensor*, *neg_dist: Tensor*, *fmr_vals: Tuple[float, ...] = (0.1,)*) → Tensor

Function to compute False Non Match Rate (FNMR) value when False Match Rate (FMR) value is equal to `fmr_vals`.

The metric calculates the quantile of positive distances higher than a given q-th quantile of negative distances.

Parameters

- **pos_dist** – Distances between relevant samples.
- **neg_dist** – Distances between non-relevant samples.
- **fmr_vals** – Values of `fmr` (measured in quantiles) to compute the corresponding `fnmr`. For example, if `fmr_values` is (0.2, 0.4) we will calculate `fnmr@fmr=0.2` and `fnmr@fmr=0.4`

Returns

Tensor of `fnmr@fmr` values.

Given a vector of N distances between relevant samples, u, the false non-match rate (FNMR) is computed as the proportion of u below some threshold, T:

$$\text{FNMR}(T) = \frac{1}{N}\sum_{i=1}^{N} H\left(u_i - T\right) = 1 - \frac{1}{N}\sum_{i=1}^{N} H\left(T - u_i\right)$$

where $H(x)$ is the unit step function, and $H(0)$ taken to be 1.

Similarly, given a vector of N distances between non-relevant samples, v, the false match rate (FMR) is computed as the proportion of v above some threshold, T:

$$\text{FMR}(T) = 1 - \frac{1}{N}\sum_{i=1}^{N} H\left(v_i - T\right) = \frac{1}{N}\sum_{i=1}^{N} H\left(T - v_i\right)$$

Given some interesting false match rate values FMR_k one can find thresholds T_k corresponding to FMR measurements

$$T_k = Q_v\left(\text{FMR}_k\right)$$

where Q is the quantile function, and evaluate the corresponding values of FNMR@FMR $(T_k) \stackrel{\text{def}}{=}$ FNMR (T_k).

See:

Biometrics Performance

BIOMETRIC RECOGNITION: A MODERN ERA FOR SECURITY

Example

```
>>> pos_dist = torch.tensor([0, 0, 1, 1, 2, 2, 5, 5, 9, 9])
>>> neg_dist = torch.tensor([3, 3, 4, 4, 6, 6, 7, 7, 8, 8])
>>> calc_fnmr_at_fmr(pos_dist, neg_dist, fmr_vals=(0.1, 0.5))
tensor([0.4000, 0.2000])
```

19.8 calc_pcf

oml.functional.metrics.**calc_pcf**(*embeddings: Tensor*, *pcf_variance: Tuple[float, ...]*) → List[Tensor]

Function estimates the Principal Components Fraction (PCF) of embeddings using Principal Component Analysis. The metric is defined as a fraction of components needed to explain the required variance in data.

Parameters

- **embeddings** – Embeddings matrix with the shape of `[n_embeddings, embeddings_dim]`.
- **pcf_variance** – Values in range [0, 1]. Find the number of components such that the amount of variance that needs to be explained is greater than the fraction specified by `pcf_variance`.

Returns

List of linear dimensions as a fractions of the embeddings dimension.

Let X be a set of d dimensional embeddings. Let $\lambda_1, \ldots, \lambda_d \in \mathbb{R}$ be a set of eigenvalues of the covariance matrix of X sorted in descending order. Then for a given value of desired explained variance r, the number of principal components that explaines $r \cdot 100\% n$ such that

$$\frac{\sum_{i=1}^{n-1} \lambda_i}{\sum_{i=1}^{d} \lambda_i} \leq r$$

The function returns

$$\frac{n}{d}$$

See:

Principal Components Analysis

Example

In the example bellow there are 4 vectors of length 10, and only first 4 dimensions have non-zero values. Its covariance matrix will have only 4 eigenvalues that are greater than 0, i.e. there are only 4 principal axes. So, in order to keep at least 50% of the information from the set, we need to keep 2 principal axes, and in order to keep all the information we need to keep 5 principal axes (one additional axis appears because the number of principal axes is superior to the desired explained variance threshold).

```
>>> embeddings = torch.eye(4, 10, dtype=torch.float)
>>> calc_pcf(embeddings, pcf_variance=(0.5, 1))
tensor([0.2000, 0.5000])
```

CHAPTER

TWENTY

PYTORCH LIGHTNING

20.1 MetricValCallback

class oml.lightning.callbacks.metric.**MetricValCallback**(*metric:* IBasicMetric, *log_images: bool = False*, *loader_idx: int = 0*, *samples_in_getitem: int = 1*)

Bases: `Callback`

This is a wrapper which allows to use IBasicMetric with PyTorch Lightning.

__init__(*metric:* IBasicMetric, *log_images: bool = False*, *loader_idx: int = 0*, *samples_in_getitem: int = 1*)

Parameters

- **metric** – Metric
- **log_images** – Set `True` if you want to have visual logging
- **loader_idx** – Idx of the loader to calculate metric for
- **samples_in_getitem** – Some of the datasets return several samples when calling `__getitem__`, so we need to handle it for the proper calculation. For most of the cases this value equals to 1, but for the dataset which explicitly return triplets, this value must be equal to 3, for a dataset of pairs it must be equal to 2.

20.2 ExtractorModule

class oml.lightning.modules.extractor.**ExtractorModule**(*extractor: IExtractor*, *criterion: Optional[Module] = None*, *optimizer: Optional[Optimizer] = None*, *scheduler: Optional[_LRScheduler] = None*, *scheduler_interval: str = 'step'*, *scheduler_frequency: int = 1*, *input_tensors_key: str = 'input_tensors'*, *labels_key: str = 'labels'*, *embeddings_key: str = 'embeddings'*, *scheduler_monitor_metric: Optional[str] = None*, *freeze_n_epochs: int = 0*)

Bases: `LightningModule`

This is a base module to train your model with Lightning.

__init__(*extractor: IExtractor*, *criterion: Optional[Module] = None*, *optimizer: Optional[Optimizer] = None*, *scheduler: Optional[_LRScheduler] = None*, *scheduler_interval: str = 'step'*, *scheduler_frequency: int = 1*, *input_tensors_key: str = 'input_tensors'*, *labels_key: str = 'labels'*, *embeddings_key: str = 'embeddings'*, *scheduler_monitor_metric: Optional[str] = None*, *freeze_n_epochs: int = 0*)

Parameters

- **extractor** – Extractor to train
- **criterion** – Criterion to optimize
- **optimizer** – Optimizer
- **scheduler** – Learning rate scheduler
- **scheduler_interval** – Interval of calling scheduler (must be `step` or `epoch`)
- **scheduler_frequency** – Frequency of calling scheduler
- **input_tensors_key** – Key to get tensors from the batches
- **labels_key** – Key to get labels from the batches
- **embeddings_key** – Key to get embeddings from the batches
- **scheduler_monitor_metric** – Metric to monitor for the schedulers that depend on the metric value
- **freeze_n_epochs** – number of epochs to freeze model (for n > 0 model has to be a successor of `IFreezable` interface). When `current_epoch >= freeze_n_epochs` model is unfreezed. Note that epochs are starting with 0.

20.3 extractor_training_pipeline

oml.lightning.pipelines.train.**extractor_training_pipeline**(*cfg: Union[Dict[str, Any], DictConfig]*) → None

This pipeline allows you to train and validate a feature extractor which represents images as feature vectors.

The config can be specified as a dictionary or with hydra: https://hydra.cc/. For more details look at `pipelines/features_extraction/README.md`

20.4 extractor_validation_pipeline

oml.lightning.pipelines.validate.**extractor_validation_pipeline**(*cfg: Union[Dict[str, Any], DictConfig]*) → Tuple[Trainer, Dict[str, Any]]

This pipeline allows you to validate a feature extractor which represents images as feature vectors.

The config can be specified as a dictionary or with hydra: https://hydra.cc/. For more details look at `pipelines/features_extraction/README.md`

20.5 extractor_prediction_pipeline

oml.lightning.pipelines.predict.**extractor_prediction_pipeline**(*cfg: Union[Dict[str, Any], DictConfig]*) → None

This pipeline allows you to save features extracted by a feature extractor.

The config can be specified as a dictionary or with hydra: https://hydra.cc/. For more details look at `pipelines/features_extraction/README.md`

20.6 postprocessor_training_pipeline

oml.lightning.pipelines.train_postprocessor.**postprocessor_training_pipeline**(*cfg: DictConfig*) → None

This pipeline allows you to train and validate a pairwise postprocessor which fixes mistakes of a feature extractor in retrieval setup.

The config can be specified as a dictionary or with hydra: https://hydra.cc/. For more details look at `pipelines/postprocessing/pairwise_postprocessing/README.md`

CHAPTER

TWENTYONE

UTILS

21.1 check_retrieval_dataframe_format

oml.utils.dataframe_format.**check_retrieval_dataframe_format**(*df: Union[Path, str, DataFrame]*, *dataset_root: Optional[Path] = None*, *sep: str = ','*, *verbose: bool = True*) → None

Function checks if the dataset is in the correct format.

Parameters

- **df** – Path to `.csv` file or pandas DataFrame
- **dataset_root** – Path to the dataset root, set `None` if you used absolute paths in your dataframe
- **sep** – Separator used in `.csv`
- **verbose** – Set `True` if you want to see warnings

21.2 download_mock_dataset

oml.utils.download_mock_dataset.**download_mock_dataset**(*dataset_root: Union[str, Path]*, *check_md5: bool = True*, *df_name: str = 'df.csv'*) → Tuple[DataFrame, DataFrame]

Function to download mock dataset which is already prepared in the required format.

Parameters

- **dataset_root** – Path to save the dataset
- **check_md5** – Set `True` to check md5sum

- **df_name** – Name of csv file for which output DataFrames will be returned

Returns: Dataframes for the training and validation stages

21.3 PCA

class oml.utils.misc_torch.**PCA**(*embeddings: Tensor*)

Bases: object

Principal component analysis (PCA).

Estimate principal axes, and perform vectors transformation.

Note: The code is almost the same as one from sklearn, but we had two reasons to have our own implementation. First, we need to work with Torch tensors instead of NumPy arrays. Second, we wanted to avoid one more external dependency.

---

components

Matrix of shape `[embeddings_dim, embeddings_dim]`. Principal axes in embeddings space, representing the directions of maximum variance in the data. Equivalently, the right singular vectors of the centered input data, parallel to its eigenvectors. The components are sorted by `explained_variance`.

Type
torch.Tensor

explained_variance

Array of size `embeddings_dim` The amount of variance explained by each of the selected components. The variance estimation uses `n_embeddings - 1` degrees of freedom. Equal to eigenvalues of the covariance matrix of `embeddings`.

Type
torch.Tensor

explained_variance_ratio

Array of size `embeddings_dim`. Percentage of variance explained by each of the components.

Type
torch.Tensor

singular_values

Array of size `embeddings_dim`. The singular values corresponding to each of the selected components.

Type
torch.Tensor

mean

Array of size `embeddings_dim`. Per-feature empirical mean, estimated from the training set. Equal to `embeddings.mean(dim=0)`.

Type
torch.Tensor

For an embeddings matrix X of shape $n \times d$ the principal axes could be found by performing Singular Value Decomposition

$$X = U \Sigma V^T$$

where U is an $n \times n$ orthogonal matrix, Σ is an $n \times d$ rectangular diagonal matrix with non-negative real numbers on the diagonal, V is an $d \times d$ orthogonal matrix.

Rows of the V form an orthonormal basis, and could be used to project embeddings to a new space, possible of lower dimension:

$$X' = X \cdot V^T$$

The inverse transform is done by

$$X = X' \cdot V$$

See:

Principal Components Analysis

Example

```
>>> embeddings = torch.rand(100, 5)
>>> pca = PCA(embeddings)
>>> embeddings_transformed = pca.transform(embeddings)
>>> embeddings_recovered = pca.inverse_transform(embeddings_transformed)
>>> torch.all(torch.isclose(embeddings, embeddings_recovered, atol=1.e-6))
tensor(True)
```

__init__(*embeddings: Tensor*)

Parameters

embeddings – Embeddings matrix with the shape of `[n_embeddings, embeddings_dim]`.

transform(*embeddings: Tensor*, *n_components: Optional[int] = None*) → Tensor

Apply fitted PCA to transform embeddings.

Parameters

- **embeddings** – Matrix of shape `[n_embeddings, embeddings_dim]`.
- **n_components** – The desired dimension of the output.

Returns

Transformed embeddings.

inverse_transform(*embeddings: Tensor*) → Tensor

Apply inverse transform to embeddings.

Parameters

embeddings – Matrix of shape `[n_embeddings, N]` where `N <= embeddings_dim` is the dimension of embeddings.

Returns

Embeddings projected into original embeddings space.

calc_principal_axes_number(*pcf_variance: Tuple[float, ...]*) → Tensor

Function estimates the number of principal axes that are required to explain the *explained_variance_ths* variance.

Parameters

pcf_variance – Values in range [0, 1]. Find the number of components such that the amount of variance that needs to be explained is greater than the fraction specified by `pcf_variance`.

Returns
List of amount of principal axes.

Let X be a set of d dimensional embeddings. Let $\lambda_1, \ldots, \lambda_d \in \mathbb{R}$ be a set of eigenvalues of the covariance matrix of X sorted in descending order. Then for a given value of desired explained variance r, the number of principal components that explaines $r \cdot 100\% n$ such that

$$\frac{\sum_{i=1}^{n-1} \lambda_i}{\sum_{i=1}^{d} \lambda_i} \leq r$$

Example

In the example bellow there are 4 vectors of length 10, and only first 4 dimensions have non-zero values. Its covariance matrix will have only 4 eigenvalues, that are greater than 0, i.e. there are only 4 principal axes. So, in order to keep at least 50% of the information from the set, we need to keep 2 principal axes, and in order to keep all the information we need to keep 5 principal axes (one additional axis appears because the number of principal axes is superior to the desired explained variance threshold).

```
>>> embeddings = torch.eye(4, 10, dtype=torch.float)
>>> pca = PCA(embeddings)
>>> pca.calc_principal_axes_number(pcf_variance=(0.5, 1))
tensor([2, 5])
```

21.4 take_2d

oml.utils.misc_torch.**take_2d**(*x: Tensor*, *indices: Tensor*) → Tensor

Parameters

- **x** – Tensor with the shape of `[N, M]`
- **indices** – Tensor of integers with the shape of `[N, P]` Note, rows in `indices` may contain duplicated values. It means that we can take the same element from `x` several times.

Returns
Tensor of the items picked from `x` with the shape of `[N, P]`

21.5 assign_2d

oml.utils.misc_torch.**assign_2d**(*x: Tensor*, *indices: Tensor*, *new_values: Tensor*) → Tensor

Parameters

- **x** – Tensor with the shape of `[N, M]`
- **indices** – Tensor of integers with the shape of `[N, P]`, where `P <= M`
- **new_values** – Tensor with the shape of `[N, P]`

Returns
`x[i, indices[i, j]] = new_values[i, j]`

Return type
Tensor with the shape of `[N, M]` constructed by the following rule

CHAPTER

TWENTYTWO

DDP

Note, that this is an advanced section for developers or curious users. Normally, you don't even need to know about the existence of the classes and functions below.

22.1 IMetricDDP

class oml.interfaces.metrics.**IMetricDDP**

Bases: *IBasicMetric*

This is an extension of a base metric interface to work in DDP mode

abstract sync() → None

Method aggregates data in DDP mode before metrics calculations

22.2 EmbeddingMetricsDDP

class oml.metrics.embeddings.**EmbeddingMetricsDDP**(*embeddings_key: str = 'embeddings', labels_key: str = 'labels', is_query_key: str = 'is_query', is_gallery_key: str = 'is_gallery', extra_keys: Tuple[str, ...] = (), cmc_top_k: Tuple[int, ...] = (5,), precision_top_k: Tuple[int, ...] = (5,), map_top_k: Tuple[int, ...] = (5,), fmr_vals: Tuple[float, ...] = (), pcf_variance: Tuple[float, ...] = (0.5,), categories_key: Optional[str] = None, sequence_key: Optional[str] = None, postprocessor: Optional[IDistancesPostprocessor] = None, metrics_to_exclude_from_visualization: Iterable[str] = (), return_only_overall_category: bool = False, visualize_only_overall_category: bool = True, verbose: bool = True*)

Bases: *EmbeddingMetrics*, *IMetricDDP*

__init__(*embeddings_key: str = 'embeddings', labels_key: str = 'labels', is_query_key: str = 'is_query', is_gallery_key: str = 'is_gallery', extra_keys: Tuple[str, ...] = (), cmc_top_k: Tuple[int, ...] = (5,), precision_top_k: Tuple[int, ...] = (5,), map_top_k: Tuple[int, ...] = (5,), fmr_vals: Tuple[float, ...] = (), pcf_variance: Tuple[float, ...] = (0.5,), categories_key: Optional[str] = None, sequence_key: Optional[str] = None, postprocessor: Optional[IDistancesPostprocessor] = None, metrics_to_exclude_from_visualization: Iterable[str] = (), return_only_overall_category: bool = False, visualize_only_overall_category: bool = True, verbose: bool = True*)

Parameters

- **embeddings_key** – Key to take the embeddings from the batches
- **labels_key** – Key to take the labels from the batches
- **is_query_key** – Key to take the information whether every batch sample belongs to the query
- **is_gallery_key** – Key to take the information whether every batch sample belongs to the gallery
- **extra_keys** – Keys to accumulate some additional information from the batches
- **cmc_top_k** – Values of `k` to calculate `cmc@k` (*Cumulative Matching Characteristic*)
- **precision_top_k** – Values of `k` to calculate `precision@k`
- **map_top_k** – Values of `k` to calculate `map@k` (*Mean Average Precision*)
- **fmr_vals** – Values of `fmr` (measured in quantiles) to calculate `fnmr@fmr` (*False Non Match Rate at the given False Match Rate*). For example, if `fmr_values` is (0.2, 0.4) we will calculate `fnmr@fmr=0.2` and `fnmr@fmr=0.4`. Note, computing this metric requires additional memory overhead, that is why it's turned off by default.
- **pcf_variance** – Values in range [0, 1]. Find the number of components such that the amount of variance that needs to be explained is greater than the percentage specified by `pcf_variance`.
- **categories_key** – Key to take the samples' categories from the batches (if you have ones)
- **sequence_key** – Key to take sequence ids from the batches (if you have ones)
- **postprocessor** – Postprocessor which applies some techniques like query reranking
- **metrics_to_exclude_from_visualization** – Names of the metrics to exclude from the visualization. It will not affect calculations.

- **return_only_overall_category** – Set `True` if you want to return only the aggregated metrics
- **visualize_only_overall_category** – Set `False` if you want to visualize each category separately
- **verbose** – Set `True` if you want to print metrics

setup(*num_samples: int*) → None

Method for preparing metrics to work: memory allocation, placeholder preparation, etc. Has to be called before the first call of `self.update_data()`.

update_data(*data_dict: Dict[str, Any]*, *indices: List[int]*) → None

Parameters

- **data_dict** – Batch of data containing records of the same size: `bs`.
- **indices** – Global indices of the elements in your records within the range of `(0, dataset_size - 1)`. Indices are needed in DDP (because data is gathered shuffled, additionally you may also get some duplicates due to padding). In the single device regime it's may be useful if you accumulate data in shuffled order.

compute_metrics() → Dict[Union[str, int], Dict[str, Dict[Union[int, float], Union[float, Tensor]]]]

The output must be in the following format:

```
{
    "self.overall_categories_key": {"metric1": ..., "metric2": ...},
    "category1": {"metric1": ..., "metric2": ...},
    "category2": {"metric1": ..., "metric2": ...}
}
```

Where `category1` and `category2` are optional.

sync() → None

Method aggregates data in DDP mode before metrics calculations

22.3 ModuleDDP

class oml.lightning.modules.ddp.**ModuleDDP**(*loaders_train: Optional[Union[DataLoader, Sequence[DataLoader], Dict[str, DataLoader]]] = None*, *loaders_val: Optional[Union[DataLoader, Sequence[DataLoader]]] = None*)

Bases: `LightningModule`

The module automatically patches training and validation dataloaders to DDP mode by splitting available indices between devices. Note, don't use `trainer.fit(...)` or `trainer.validate(...)`, because in this case, *PytorchLightning* will ignore our patching.

__init__(*loaders_train: Optional[Union[DataLoader, Sequence[DataLoader], Dict[str, DataLoader]]] = None*, *loaders_val: Optional[Union[DataLoader, Sequence[DataLoader]]] = None*)

22.4 ExtractorModuleDDP

class oml.lightning.modules.extractor.**ExtractorModuleDDP**(*loaders_train: Optional[Any] = None*, *loaders_val: Optional[Any] = None*, **args: Any*, ***kwargs: Any*)

Bases: *ExtractorModule*, *ModuleDDP*

This is a base module for the training of your model with Lightning in DDP.

__init__(*loaders_train: Optional[Any] = None*, *loaders_val: Optional[Any] = None*, **args: Any*, ***kwargs: Any*)

Parameters

- **extractor** – Extractor to train
- **criterion** – Criterion to optimize
- **optimizer** – Optimizer
- **scheduler** – Learning rate scheduler
- **scheduler_interval** – Interval of calling scheduler (must be `step` or `epoch`)
- **scheduler_frequency** – Frequency of calling scheduler
- **input_tensors_key** – Key to get tensors from the batches
- **labels_key** – Key to get labels from the batches
- **embeddings_key** – Key to get embeddings from the batches
- **scheduler_monitor_metric** – Metric to monitor for the schedulers that depend on the metric value
- **freeze_n_epochs** – number of epochs to freeze model (for n > 0 model has to be a successor of `IFreezable` interface). When `current_epoch >= freeze_n_epochs` model is unfreezed. Note that epochs are starting with 0.

22.5 MetricValCallbackDDP

class oml.lightning.callbacks.metric.**MetricValCallbackDDP**(*metric:* IMetricDDP, **args: Any*, ***kwargs: Any*)

Bases: *MetricValCallback*

This is an extension to the regular callback that takes into account data reduction and padding on the inference for each device in DDP setup

__init__(*metric:* IMetricDDP, **args: Any*, ***kwargs: Any*)

Parameters

- **metric** – Metric
- **log_images** – Set `True` if you want to have visual logging
- **loader_idx** – Idx of the loader to calculate metric for

- **samples_in_getitem** – Some of the datasets return several samples when calling `__getitem__`, so we need to handle it for the proper calculation. For most of the cases this value equals to 1, but for the dataset which explicitly return triplets, this value must be equal to 3, for a dataset of pairs it must be equal to 2.

22.6 DDPSamplerWrapper

class oml.ddp.patching.**DDPSamplerWrapper**(*sampler: Union[BatchSampler, Sampler,* IBatchSampler*]*, *shuffle_samples_between_gpus: bool = True*, *pad_data_to_num_gpus: bool = True*)

Bases: `DistributedSampler`

This is a wrapper to allow using custom sampler in DDP mode.

Default *DistributedSampler* allows us to build a sampler for a dataset in DDP mode. Usually we can easily replace default *SequentialSampler* [when `DataLoader(shuffle=False, ...)`] and *RandomSampler* [when `DataLoader(shuffle=True, ...)`] with *DistributedSampler*. But for the custom sampler, we need an extra wrapper.

Thus, this wrapper distributes indices produced by sampler among several devices for further usage.

__init__(*sampler: Union[BatchSampler, Sampler,* IBatchSampler*]*, *shuffle_samples_between_gpus: bool = True*, *pad_data_to_num_gpus: bool = True*)

Parameters

- **sampler** – Sequential or batch sampler
- **pad_data_to_num_gpus** – When using DDP we should manage behavior with the last batch, because each device should have the same amount of data. If the sampler length is not evenly divisible by the number of devices, we must duplicate part of the data (`pad_data_to_num_gpus=True`), or discard part of the data (`pad_data_to_num_gpus=False`).
- **shuffle_samples_between_gpus** – shuffle available indices before feeding them to GPU. Note, that shuffle inside GPU after the feeding will be used according to behavior of the sampler.

Note: Wrapper can be used with both the default *SequentialSampler* or *RandomSampler* from *PyTorch* and with some custom sampler.

_reload() → None

We need to re-instantiate the wrapper in order to update the available indices for the new epoch. We don't perform this step on the epoch 0, because we want to be comparable with no DDP setup there.

22.7 patch_dataloader_to_ddp

oml.ddp.patching.**patch_dataloader_to_ddp**(*loader: DataLoader*) → DataLoader

Function inspects loader and modifies sampler for working in DDP mode.

Note: We ALWAYS use the padding of samples (in terms of the number of batches or number of samples per epoch) in order to use the same amount of data for each device in DDP. Thus, the behavior with and without DDP may be slightly different (e.g. metrics values).

22.8 sync_dicts_ddp

oml.ddp.utils.**sync_dicts_ddp**(*outputs_from_device: Dict[str, Any]*, *world_size: int*, *device: Union[device, str] = 'cpu'*) → Dict[str, Any]

The function allows you to combine and merge data stored in dictionaries from all devices. You can place this function in your code and all the devices upon reaching this function will wait for each other to synchronize and merge dictionaries.

Note: The function under the hood pickles all object, converts bytes to tensor, then unpickles them after syncing. With *NCCL DDP* backend (the default one) intermediate tensors are stored on CUDA.

CHAPTER

TWENTYTHREE

RETRIEVAL POST-PROCESSING

23.1 IDistancesPostprocessor

class oml.interfaces.retrieval.**IDistancesPostprocessor**

Bases: `object`

This is a parent class for the classes which apply some postprocessing after query-to-gallery distance matrix has been calculated. For example, we may want to apply one of re-ranking techniques.

process(*distances: Tensor*, *queries: Any*, *galleries: Any*) → Tensor

This method takes all the needed variables and returns the modified matrix of distances, where some distances are replaced with new ones.

Parameters

- **distances** – Matrix with the shape of `[Q, G]`
- **queries** – Queries in the amount of `Q`
- **galleries** – Galleries in the amount of `G`

Returns

An updated distances matrix with the shape of `[Q, G]`

23.2 PairwisePostprocessor

class oml.retrieval.postprocessors.pairwise.**PairwisePostprocessor**

Bases: *IDistancesPostprocessor*, `ABC`

This postprocessor allows us to re-estimate the distances between queries and `top-n` galleries closest to them. It creates pairs of queries and galleries and feeds them to a pairwise model.

process(*distances: Tensor*, *queries: Any*, *galleries: Any*) → Tensor

Parameters

- **distances** – Matrix with the shape of `[Q, G]`
- **queries** – Queries in the amount of `Q`
- **galleries** – Galleries in the amount of `G`

Returns

Distance matrix with the shape of `[Q, G]`,
where `top_n` minimal values in each row have been updated by the pairwise model, other distances are shifted by a margin to keep the relative order.

inference(*queries: Any*, *galleries: Any*, *ii_top: Tensor*, *top_n: int*) → Tensor

Depends on the exact types of queries/galleries this method may be implemented differently.

Parameters

- **queries** – Queries in the amount of `Q`
- **galleries** – Galleries in the amount of `G`
- **ii_top** – Indices of the closest galleries with the shape of `[Q, top_n]`
- **top_n** – Number of the closest galleries to re-rank

Returns
An updated distance matrix with the shape of `[Q, G]`

23.3 PairwiseEmbeddingsPostprocessor

class oml.retrieval.postprocessors.pairwise.**PairwiseEmbeddingsPostprocessor**(*top_n: int*, *pairwise_model: IPairwiseModel*, *num_workers: int*, *batch_size: int*, *verbose: bool = False*, *use_fp16: bool = False*, *is_query_key: str = 'is_query'*, *is_gallery_key: str = 'is_gallery'*, *embeddings_key: str = 'embeddings'*)

Bases: *PairwisePostprocessor*

__init__(*top_n: int*, *pairwise_model: IPairwiseModel*, *num_workers: int*, *batch_size: int*, *verbose: bool = False*, *use_fp16: bool = False*, *is_query_key: str = 'is_query'*, *is_gallery_key: str = 'is_gallery'*, *embeddings_key: str = 'embeddings'*)

Parameters

- **top_n** – Model will be applied to the `num_queries * top_n` pairs formed by each query and `top_n` most relevant galleries.
- **pairwise_model** – Model which is able to take two embeddings as inputs and estimate the *distance* (not in a strictly mathematical sense) between them.
- **num_workers** – Number of workers in DataLoader
- **batch_size** – Batch size that will be used in DataLoader
- **verbose** – Set `True` if you want to see progress bar for an inference
- **use_fp16** – Set `True` if you want to use half precision
- **is_query_key** – Key to access a binary mask indicates queries in case of using `process_by_dict`
- **is_gallery_key** – Key to access a binary mask indicates galleries in case of using `process_by_dict`
- **embeddings_key** – Key to access embeddings in case of using `process_by_dict`

inference(*queries: Tensor*, *galleries: Tensor*, *ii_top: Tensor*, *top_n: int*) → Tensor

Parameters

- **queries** – Queries representations with the shape of `[Q, *]`
- **galleries** – Galleries representations with the shape of `[G, *]`
- **ii_top** – Indices of the closest galleries with the shape of `[Q, top_n]`
- **top_n** – Number of the closest galleries to re-rank

Returns

Updated distance matrix with the shape of `[Q, G]`

23.4 PairwiseImagesPostprocessor

class oml.retrieval.postprocessors.pairwise.**PairwiseImagesPostprocessor**(*top_n: int*, *pairwise_model: IPairwiseModel*, *transforms: Union[Compose, Compose]*, *num_workers: int = 0*, *batch_size: int = 128*, *verbose: bool = True*, *use_fp16: bool = False*, *is_query_key: str = 'is_query'*, *is_gallery_key: str = 'is_gallery'*, *paths_key: str = 'paths'*)

Bases: *PairwisePostprocessor*

__init__(*top_n: int*, *pairwise_model: IPairwiseModel*, *transforms: Union[Compose, Compose]*, *num_workers: int = 0*, *batch_size: int = 128*, *verbose: bool = True*, *use_fp16: bool = False*, *is_query_key: str = 'is_query'*, *is_gallery_key: str = 'is_gallery'*, *paths_key: str = 'paths'*)

Parameters

- **top_n** – Model will be applied to the `num_queries * top_n` pairs formed by each query and its `top_n` most relevant galleries.
- **pairwise_model** – Model which is able to take two images as inputs and estimate the *distance* (not in a strictly mathematical sense) between them.
- **transforms** – Transforms that will be applied to an image
- **num_workers** – Number of workers in DataLoader
- **batch_size** – Batch size that will be used in DataLoader
- **verbose** – Set `True` if you want to see progress bar for an inference
- **use_fp16** – Set `True` if you want to use half precision
- **is_query_key** – Key to access a binary mask indicates queries in case of using `process_by_dict`
- **is_gallery_key** – Key to access a binary mask indicates galleries in case of using `process_by_dict`
- **paths_key** – Key to access paths to images in case of using `process_by_dict`

inference(*queries: List[Path]*, *galleries: List[Path]*, *ii_top: Tensor*, *top_n: int*) → Tensor

Parameters

- **queries** – Paths to queries with the length of `Q`
- **galleries** – Paths to galleries with the length of `G`
- **ii_top** – Indices of the closest galleries with the shape of `[Q, top_n]`
- **top_n** – Number of the closest galleries to re-rank

Returns

Updated distance matrix with the shape of `[Q, G]`

INDEX

Symbols

A

B

C

D

E

F

G

H

I

L