Catalyst Documentation

Release 20.05

Scitator

May 08, 2020

Overview

1 Getting started **3**

2 Overview **5**
- 2.1 Installation . . . 5
- 2.2 Features . . . 5
- 2.3 Structure . . . 6
- 2.4 Tests . . . 6
- 2.5 Tutorials . . . 6

3 Community **7**
- 3.1 Contribution guide . . . 7
- 3.2 User feedback . . . 7
- 3.3 Citation . . . 7

4 Indices and tables **9**
- 4.1 Examples . . . 9
- 4.2 Distributed training . . . 10
- 4.3 Contribution . . . 15
- 4.4 Core . . . 16
- 4.5 DL . . . 18
- 4.6 Data . . . 18
- 4.7 Utilities . . . 20
- 4.8 Contrib . . . 25

Python Module Index **27**

Index **29**



PyTorch framework for Deep Learning research and development. It was developed with a focus on reproducibility, fast experimentation and code/ideas reusing. Being able to research/develop something new, rather than write another regular train loop.

Break the cycle - use the Catalyst!

Project manifest. Part of PyTorch Ecosystem. Part of Catalyst Ecosystem:

- Alchemy - Experiments logging & visualization
- Catalyst - Accelerated DL R&D
- Reaction - Convenient DL serving

Catalyst at AI Landscape.

CHAPTER 1

Getting started

```
import os
import torch
from torch.nn import functional as F
from torch.utils.data import DataLoader
from torchvision.datasets import MNIST
from torchvision.transforms import ToTensor
from catalyst import dl
from catalyst.utils import metrics

model = torch.nn.Linear(28 * 28, 10)
optimizer = torch.optim.Adam(model.parameters(), lr=0.02)

loaders = {
    "train": DataLoader(MNIST(os.getcwd(), train=True, download=True,␣
↪transform=ToTensor()), batch_size=32),
    "valid": DataLoader(MNIST(os.getcwd(), train=False, download=True,␣
↪transform=ToTensor()), batch_size=32),
}

class CustomRunner(dl.Runner):

    def predict_batch(self, batch):
        # model inference step
        return self.model(batch[0].to(self.device).view(batch[0].size(0), -1))

    def _handle_batch(self, batch):
        # model train/valid step
        x, y = batch
        y_hat = self.model(x.view(x.size(0), -1))

        loss = F.cross_entropy(y_hat, y)
        accuracy01, accuracy03 = metrics.accuracy(y_hat, y, topk=(1, 3))
        self.state.batch_metrics.update(
            {"loss": loss, "accuracy01": accuracy01, "accuracy03": accuracy03}
```

(continues on next page)

(continued from previous page)

```
        )

        if self.state.is_train_loader:
            loss.backward()
            self.state.optimizer.step()
            self.state.optimizer.zero_grad()

runner = CustomRunner()
# model training
runner.train(
    model=model,
    optimizer=optimizer,
    loaders=loaders,
    logdir="./logs",
    num_epochs=5,
    verbose=True,
    load_best_on_end=True,
)
# model inference
for prediction in runner.predict_loader(loader=loaders["valid"]):
    assert prediction.detach().cpu().numpy().shape[-1] == 10
# model tracing
traced_model = runner.trace(loader=loaders["valid"])
```

- Customizing what happens in train
- Demo with minimal examples for ML, CV, NLP, GANs and RecSys
- For Catalyst.RL introduction, please follow Catalyst.RL repo.

CHAPTER 2

Overview

Catalyst helps you write compact but full-featured Deep Learning pipelines in a few lines of code. You get a training loop with metrics, early-stopping, model checkpointing and other features without the boilerplate.

2.1 Installation

Common installation:

```
pip install -U catalyst
```

More specific with additional requirements:

```
pip install catalyst[ml]          # installs DL+ML based catalyst
pip install catalyst[cv]          # installs DL+CV based catalyst
pip install catalyst[nlp]         # installs DL+NLP based catalyst
pip install catalyst[ecosystem]   # installs Catalyst.Ecosystem
pip install catalyst[contrib]     # installs DL+contrib based catalyst
pip install catalyst[all]         # installs everything
# and master version installation
pip install git+https://github.com/catalyst-team/catalyst@master --upgrade
```

Catalyst is compatible with: Python 3.6+. PyTorch 1.0.0+.

2.2 Features

- Universal train/inference loop.
- Configuration files for model/data hyperparameters.
- Reproducibility – all source code and environment variables will be saved.
- Callbacks – reusable train/inference pipeline parts with easy customization.

- Training stages support.
- Deep Learning best practices - SWA, AdamW, Ranger optimizer, OneCycle, and more.
- Developments best practices - fp16 support, distributed training, slurm.

2.3 Structure

- **contrib** - additional modules contributed by Catalyst users.
- **core** - framework core with main abstractions - Experiment, Runner, Callback and State.
- **data** - useful tools and scripts for data processing.
- **dl** – runner for training and inference, all of the classic ML and CV/NLP/RecSys metrics and a variety of callbacks for training, validation and inference of neural networks.
- **utils** - typical utils for Deep Learning research.

2.4 Tests

All the Catalyst code is tested rigorously with every new PR.

In fact, we train a number of different models for various of tasks - image classification, image segmentation, text classification, GAN training. During the tests, we compare their convergence metrics in order to verify the correctness of the training procedure and its reproducibility.

Overall, Catalyst guarantees fully tested, correct and reproducible best practices for the automated parts.

2.5 Tutorials

- Demo with minimal examples for ML, CV, NLP, GANs and RecSys
- Detailed classification tutorial
- Advanced segmentation tutorial
- Comprehensive classification pipeline
- Binary and semantic segmentation pipeline
- Beyond fashion: Deep Learning with Catalyst (Config API)
- Tutorial from Notebook API to Config API (RU)

In the examples of the repository, you can find advanced tutorials and Catalyst best practices.

CHAPTER 3

Community

3.1 Contribution guide

We appreciate all contributions. If you are planning to contribute back bug-fixes, please do so without any further discussion. If you plan to contribute new features, utility functions or extensions, please first open an issue and discuss the feature with us.

Please see the contribution guide for more information.

By participating in this project, you agree to abide by its Code of Conduct.

3.2 User feedback

We have created catalyst.team.core@gmail.com for "user feedback".

- If you like the project and want to say thanks, this the right place.
- If you would like to start a collaboration between your team and Catalyst team to do better Deep Learning R&D - you are always welcome.
- If you just don't like Github issues and this ways suits you better - feel free to email us.
- Finally, if you do not like something, please, share it with us and we can see how to improve it.

We appreciate any type of feedback. Thank you!

3.3 Citation

Please use this bibtex if you want to cite this repository in your publications:

```
@misc{catalyst,
    author = {Kolesnikov, Sergey},
    title = {Accelerated DL R&D},
    year = {2018},
    publisher = {GitHub},
    journal = {GitHub repository},
    howpublished = {\url{https://github.com/catalyst-team/catalyst}},
}
```

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

4.1 Examples

4.1.1 Tutorials

1. **classification tutorial**
    - dataset preparation (raw images -> train/valid/infer splits)
    - augmentations usage example
    - pretrained model finetuning
    - various classification metrics
    - metrics visualizaiton
    - FocalLoss and OneCycle usage examples
    - class imbalance handling
    - model inference
2. **segmentation tutorial**
    - car segmentation dataset
    - augmentations with albumentations library
    - training in FP16 with NVIDIA Apex
    - using segmentation models from `catalyst/contrib/models/cv/segmentation`

- training with multiple criterion (Dice + IoU + BCE) example
- Lookahead + RAdam optimizer usage example
- tensorboard logs visualization
- predictions visualization
- Test-time augmentations with ttach library

4.1.2 Pipelines

1. **Full description of configs with comments:**
    - Eng
    - Rus
2. **classification pipeline**
    - classification model training and inference
    - different augmentations and stages usage
    - metrics visualization with tensorboard
3. **segmentation pipeline**
    - binary and semantic segmentation with U-Net
    - model training and inference
    - different augmentations and stages usage
    - metrics visualization with tensorboard

4.1.3 RL tutorials & pipelines

For Reinforcement Learning examples check out our Catalyst.RL repo.

4.2 Distributed training

If you have multiple GPUs, the most reliable way to use all of them for training is to use the distributed package from pytorch. To help you, there is a distributed helpers in Catalyst to make it really easy.

Note, that current distributed implementation requires you to run only training procedure in your python scripts.

4.2.1 Prepare your script

Distributed training doesn't work in a notebook, so prepare a script to run the training. For instance, here is a minimal script that trains a linear regression model.

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

from catalyst.dl import SupervisedRunner
```

(continues on next page)

(continued from previous page)

```
# experiment setup
logdir = "./logdir"
num_epochs = 8

# data
num_samples, num_features = int(1e4), int(1e1)
X, y = torch.rand(num_samples, num_features), torch.rand(num_samples)
dataset = TensorDataset(X, y)
loader = DataLoader(dataset, batch_size=32, num_workers=1)
loaders = {"train": loader, "valid": loader}

# model, criterion, optimizer, scheduler
model = torch.nn.Linear(num_features, 1)
criterion = torch.nn.MSELoss()
optimizer = torch.optim.Adam(model.parameters())
scheduler = torch.optim.lr_scheduler.MultiStepLR(optimizer, [3, 6])

# model training
runner = SupervisedRunner()
runner.train(
    model=model,
    criterion=criterion,
    optimizer=optimizer,
    scheduler=scheduler,
    loaders=loaders,
    logdir=logdir,
    num_epochs=num_epochs,
    verbose=True,
)
```

Link to the projector script.

4.2.2 Stage 1 - I just want distributed

In case you want to run it fast and ugly, with minimal changes, you can just pass `distributed=True` to `.train` call

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

from catalyst.dl import SupervisedRunner

# data
num_samples, num_features = int(1e4), int(1e1)
X, y = torch.rand(num_samples, num_features), torch.rand(num_samples)
dataset = TensorDataset(X, y)
loader = DataLoader(dataset, batch_size=32, num_workers=1)
loaders = {"train": loader, "valid": loader}

# model, criterion, optimizer, scheduler
model = torch.nn.Linear(num_features, 1)
criterion = torch.nn.MSELoss()
optimizer = torch.optim.Adam(model.parameters())
scheduler = torch.optim.lr_scheduler.MultiStepLR(optimizer, [3, 6])
```

(continues on next page)

(continued from previous page)

```
# model training
runner = SupervisedRunner()
runner.train(
    model=model,
    criterion=criterion,
    optimizer=optimizer,
    scheduler=scheduler,
    loaders=loaders,
    logdir="./logs/example_1",
    num_epochs=8,
    verbose=True,
    distributed=True,
)
```

Link to the stage-1 script.

In this way Catalyst will try to automatically make your loaders work in distributed setup and will run experiment training.

Nevertheless it has several disadvantages,

- you create your loader again and again with each distributed worker, +1 for master scripts with all processes joined.
- you can't understand what is going under the hood of `distributed=True`
- we can't always transfer your loaders to distributed mode correctly

4.2.3 Case 2 - We are going deeper

Let's make it more reusable:

```
import torch
from torch.utils.data import TensorDataset

from catalyst.dl import SupervisedRunner

# data
num_samples, num_features = int(1e4), int(1e1)
X = torch.rand(int(1e4), num_features)
y = torch.rand(X.shape[0])
dataset = TensorDataset(X, y)

# model, criterion, optimizer, scheduler
model = torch.nn.Linear(num_features, 1)
criterion = torch.nn.MSELoss()
optimizer = torch.optim.Adam(model.parameters())
scheduler = torch.optim.lr_scheduler.MultiStepLR(optimizer, [3, 6])

runner = SupervisedRunner()
runner.train(
    model=model,
    datasets={
        "batch_size": 32,
        "num_workers": 1,
        "train": dataset,
        "valid": dataset,
```

(continues on next page)

(continued from previous page)

```
    },
    criterion=criterion,
    optimizer=optimizer,
    logdir="./logs/example_2",
    num_epochs=8,
    verbose=True,
    distributed=True,
)
```

Link to the stage-2 script.

By this way we easily can transfer your datasets to distributed mode. But again, you recreate your dataset with each worker. Can we make it better?

4.2.4 Case 3 - Best practices for distributed training

Yup, check this one, distributed training like a pro:

```
import torch
from torch.utils.data import TensorDataset

from catalyst.dl import SupervisedRunner, utils

def datasets_fn(num_features: int):
    X = torch.rand(int(1e4), num_features)
    y = torch.rand(X.shape[0])
    dataset = TensorDataset(X, y)
    return {"train": dataset, "valid": dataset}

def train():
    num_features = int(1e1)
    # model, criterion, optimizer, scheduler
    model = torch.nn.Linear(num_features, 1)
    criterion = torch.nn.MSELoss()
    optimizer = torch.optim.Adam(model.parameters())
    scheduler = torch.optim.lr_scheduler.MultiStepLR(optimizer, [3, 6])

    runner = SupervisedRunner()
    runner.train(
        model=model,
        datasets={
            "batch_size": 32,
            "num_workers": 1,
            "get_datasets_fn": datasets_fn,
            "num_features": num_features,
        },
        criterion=criterion,
        optimizer=optimizer,
        scheduler=scheduler,
        logdir="./logs/example_3",
        num_epochs=8,
        verbose=True,
        distributed=False,
    )

utils.distributed_cmd_run(train)
```

Link to the stage-3 script.

Advantages,

- you have control about what is going on with manual call of `utils.distributed_cmd_run`.
- you don't duplicate the data - it calls when it really needed
- we still can easily transfer them to distributed mode, thanks to `Datasets` usage

4.2.5 Launch your training

In your terminal, type the following line (adapt *script_name* to your script name ending with .py).

```
python {script_name}
```

You can vary availble GPUs with `CUDA_VIBIBLE_DEVICES` option, for example,

```
# run only on 1st and 2nd GPUs
CUDA_VISIBLE_DEVICES="1,2" python {script_name}
```

```
# run only on 0, 1st and 3rd GPUs
CUDA_VISIBLE_DEVICES="0,1,3" python {script_name}
```

What will happen is that the same model will be copied on all your available GPUs. During training, the full dataset will randomly be split between the GPUs (that will change at each epoch). Each GPU will grab a batch (on that fractioned dataset), pass it through the model, compute the loss then back-propagate the gradients. Then they will share their results and average them, which means like your training is the equivalent of a training with a batch size of `` `batch_size x num_gpus `` (where `batch_size` is what you used in your script).

Since they all have the same gradients at this stage, they will al perform the same update, so the models will still be the same after this step. Then training continues with the next batch, until the number of desired iterations is done.

During training Catalyst will automatically average all metrics and log them on `Master` node only. Same logic used for model checkpointing.

4.2.6 Slurm support

Catalyst supports distributed training of neural networks on HPC under slurm control. Catalyst automatically allocates roles between nodes and syncs them. This allows to run experiments without any changes in the configuration file or model code. We recommend using nodes with the same number and type of GPU. You can run the experiment with the following command:

```
# Catalyst Notebook API
srun -N 2 --gres=gpu:3 --exclusive --mem=256G python run.py
# Catalyst Config API
srun -N 2 --gres=gpu:3 --exclusive --mem=256G catalyst-dl run -C config.yml
```

In this command, we request two nodes with 3 GPUs on each node in exclusive mode, i.e. we request all available CPUs on the nodes. Each node will be allocated 256G. Note that specific startup parameters using `srun` may change depending on the specific cluster and slurm settings. For more fine-tuning, we recommend reading the slurm documentation.

4.3 Contribution

4.3.1 Issues

We use GitHub issues for bug reports and feature requests.

Step-by-step guide

New feature

1. Make an issue with your feature description;
2. We shall discuss the design and its implementation details;
3. Once we agree that the plan looks good, go ahead and implement it.

Bugfix

1. Goto GitHub issues;
2. Pick an issue and comment on the task that you want to work on this feature;
3. If you need more context on a specific issue, please ask, and we will discuss the details.

Once you finish implementing a feature or bugfix, please send a Pull Request.

If you are not familiar with creating a Pull Request, here are some guides:

- http://stackoverflow.com/questions/14680711/how-to-do-a-github-pull-request
- https://help.github.com/articles/creating-a-pull-request/

Contribution best practices

1. Install requirements

```
brew install bash # for MacOS users
pip install -r requirements/requirements.txt -r requirements/requirements-dev.txt
```

2. Break your work into small, single-purpose updates if possible. It's much harder to merge in a large change with a lot of disjoint features.
3. Submit the update as a GitHub pull request against the *master* branch.
4. Make sure that you provide docstrings for all your new methods and classes.
5. Add new unit tests for your code.
6. Check the codestyle
7. Make sure that your code passes the unit tests

Codestyle

Do not forget to check the codestyle for your PR with

```
catalyst-make-codestyle && catalyst-check-codestyle
```

Make sure to have your python packages complied with *requirements/requirements.txt* and *requirements/requirements-dev.txt* to get codestyle run clean.

Unit tests

Do not forget to check that your code passes the unit tests

```
pytest .
```

4.3.2 Documentation

Catalyst uses Google style for formatting docstrings. Length of line inside docstrings block must be limited to 80 characters to fit into Jupyter documentation popups.

Check that you have written working docs

```
make check-docs
```

The command requires Sphinx and some sphinx-specific libraries. If you don't want to install them, you may make a catalyst-dev container

```
make docker-dev
docker run \\
    -v `pwd`/:/workspace/ \\
    catalyst-dev:latest \\
    bash -c "make check-docs"
```

To build docs add environment variable REMOVE_BUILDS=0

```
REMOVE_BUILDS=0 make check-docs
```

or through docker

```
docker run \\
    -v `pwd`/:/workspace/ \\
    catalyst-dev:latest \\
    bash -c "REMOVE_BUILDS=0 make check-docs"
```

The docs will be stored in *builds/* folder.

4.4 Core

- *Core*
  - *Experiment*
  - *Runner*
  - *Callback*
  - *State*
- *Callbacks*
  - *Checkpoint*
  - *Criterion*
  - *Early Stop*
  - *Exception*
  - *Logging*
  - *Metrics*
  - *Optimizer*
  - *Scheduler*
  - *Timer*
  - *Validation*
- *Registry*
- *Utils*

4.4.1 Core

Experiment

Runner

Callback

State

4.4.2 Callbacks

Checkpoint

Criterion

Early Stop

Exception

Logging

Metrics

Optimizer

Scheduler

Timer

Validation

4.4.3 Registry

4.4.4 Utils

4.5 DL

- *Experiment*
- *Runner*
- *Callbacks*
  - *Metrics*
- *Utils*
- *Registry*

4.5.1 Experiment

4.5.2 Runner

4.5.3 Callbacks

Metrics

4.5.4 Utils

4.5.5 Registry

4.6 Data

Data subpackage has data preprocessers and dataloader abstractions.

4.6.1 Scripts

You can use scripts typing *catalyst-data* in your terminal. For example:

```
$ catalyst-data tag2label --help
```

4.6.2 Augmentor

class catalyst.data.augmentor.**Augmentor**(*dict_key: str*, *augment_fn: Callable*, *input_key: str = None*, *output_key: str = None*, ***kwargs*)

Augmentation abstraction to use with data dictionaries.

__init__(*dict_key: str*, *augment_fn: Callable*, *input_key: str = None*, *output_key: str = None*, ***kwargs*)

Parameters

- **dict_key** (*str*) – key to transform
- **augment_fn** (*Callable*) – augmentation function to use
- **input_key** (*str*) – `augment_fn` input key
- **output_key** (*str*) – `augment_fn` output key
- ****kwargs** – default kwargs for augmentations function

class catalyst.data.augmentor.**AugmentorCompose**(*key2augment_fn: Dict[str, Callable]*)

Compose augmentors.

__init__(*key2augment_fn: Dict[str, Callable]*)

Parameters **key2augment_fn** (*Dict[str, Callable]*) – mapping from input key to augmentation function to apply

class catalyst.data.augmentor.**AugmentorKeys**(*dict2fn_dict: Union[Dict[str, str], List[str]]*, *augment_fn: Callable*)

Augmentation abstraction to match input and augmentations keys.

__init__(*dict2fn_dict: Union[Dict[str, str], List[str]]*, *augment_fn: Callable*)

Parameters

- **dict2fn_dict** (*Dict[str, str]*) – keys matching dict `{input_key: augment_fn_key}`. For example: `{"image": "image", "mask": "mask"}`
- **augment_fn** – augmentation function

4.6.3 Collate Functions

4.6.4 Dataset

4.6.5 Reader

Readers are the abstraction for your dataset. They can open an elem from the dataset and transform it to data, needed by your network. For example open image by path, or read string and tokenize it.

4.6.6 Sampler

4.7 Utilities

- *Utils*
  - *Checkpoint*
  - *Config*
  - *Distributed*
  - *Hash*
  - *Initialization*
  - *Misc*
  - *Numpy*
  - *Parser*
  - *Scripts*
  - *Seed*
  - *Sys*
  - *Torch*
- *Tools*
  - *Frozen Class*
  - *Registry*
  - *Time Manager*
  - *Typing*
- *Metrics*
  - *Accuracy*
  - *Dice*
  - *F1 score*
  - *Focal*
  - *IoU*
- *Meters*
  - *Meter*
  - *AP Meter*
  - *AUC Meter*
  - *Average Value Meter*
  - *Class Error Meter*
  - *Confusion Meter*

- *Map Meter*
- *Moving Average Value Meter*
- *MSE Meter*
- *Precision-Recall-F1 Meter*

4.7.1 Utils

Checkpoint

Config

Distributed

Hash

Initialization

Misc

Numpy

Parser

Scripts

Seed

Sys

Torch

4.7.2 Tools

Frozen Class

Frozen class. Example of usage can be found in `catalyst.core.state.State`.

class `catalyst.tools.frozen_class.`**FrozenClass**

Bases: `object`

Class which prohibit `__setattr__` on existing attributes.

Examples

```
>>> class State(FrozenClass):
```

Registry

Registry. .. todo:: Representative docstring for this module

class catalyst.tools.registry.**Registry**(*default_name_key: str, default_meta_factory: Callable[[Union[Type[CT_co], Callable[[...], Any]], Tuple, Mapping[KT, VT_co]], Any] = <function _default_meta_factory>*)

Bases: collections.abc.MutableMapping

Universal class allowing to add and access various factories by name.

__init__(*default_name_key: str, default_meta_factory: Callable[[Union[Type[CT_co], Callable[[...], Any]], Tuple, Mapping[KT, VT_co]], Any] = <function _default_meta_factory>*)

Parameters

- **default_name_key** (*str*) – Default key containing factory name when creating from config
- **default_meta_factory** (*MetaFactory*) – default object that calls factory. Optional. Default just calls factory.

add(*factory: Union[Type[CT_co], Callable[[...], Any]] = None, *factories, name: str = None, **named_factories*) → Union[Type[CT_co], Callable[[...], Any]]

Adds factory to registry with it's __name__ attribute or provided name. Signature is flexible.

Parameters

- **factory** – Factory instance
- **factories** – More instances
- **name** – Provided name for first instance. Use only when pass single instance.
- **named_factories** – Factory and their names as kwargs

Returns First factory passed

Return type (Factory)

add_from_module(*module*, *prefix: Union[str, List[str]] = None*) → None

Adds all factories present in module. If __all__ attribute is present, takes ony what mentioned in it.

Parameters

- **module** – module to scan
- **prefix** (*Union[str, List[str]]*) – prefix string for all the module's factories. If prefix is a list, all values will be treated as aliases.

all() → List[str]

Returns list of names of registered items

get(*name: str*) → Union[Type[CT_co], Callable[[...], Any], None]

Retrieves factory, without creating any objects with it or raises error.

Parameters **name** – factory name

Returns factory by name

Return type Factory

get_from_params(*, *meta_factory=None*, ***kwargs*) → Union[Any, Tuple[Any, Mapping[str, Any]]]

Creates instance based in configuration dict with `instantiation_fn`. If `config[name_key]` is None, None is returned.

Parameters

- **meta_factory** – Function that calls factory the right way. If not provided, default is used.
- ****kwargs** – additional kwargs for factory

Returns result of calling `instantiate_fn(factory, **config)`

get_if_str(*obj: Union[str, Type[CT_co], Callable[[...], Any]]*)

Returns object from the registry if `obj` type is string.

get_instance(*name: str*, **args*, *meta_factory=None*, ***kwargs*)

Creates instance by calling specified factory with `instantiate_fn`.

Parameters

- **name** – factory name
- **meta_factory** – Function that calls factory the right way. If not provided, default is used
- **args** – args to pass to the factory
- ****kwargs** – kwargs to pass to the factory

Returns created instance

late_add(*cb: Callable[[Registry], None]*)

Allows to prevent cycle imports by delaying some imports till next registry query.

Parameters **cb** – Callback receives registry and must call it's methods to register factories

len() → int

Returns length of registered items

exception catalyst.tools.registry.**RegistryException**(*message*)

Bases: `Exception`

Exception class for all registry errors.

__init__(*message*)

Init.

Parameters **message** – exception message

Time Manager

Simple timer.

class catalyst.tools.time_manager.**TimeManager**

Bases: `object`

@TODO: Docs. Contribution is welcome.

__init__()

@TODO: Docs. Contribution is welcome.

reset() → None
Reset all previous timers.

start(*name: str*) → None
Starts timer `name`.

Parameters **name** (*str*) – name of a timer

stop(*name: str*) → None
Stops timer `name`.

Parameters **name** (*str*) – name of a timer

Typing

4.7.3 Metrics

Accuracy

Dice

F1 score

Focal

IoU

4.7.4 Meters

The meters from torchnet.meters.

Every meter implements `catalyst.utils.meters.meter.Meter` interface.

Meter

AP Meter

AUC Meter

Average Value Meter

Class Error Meter

Confusion Meter

Map Meter

Moving Average Value Meter

MSE Meter

Precision-Recall-F1 Meter

4.8 Contrib

- *DL*
  - *Callbacks*
- *NN*
  - *Criterion*
  - *Modules*
  - *Optimizers*
  - *Schedulers*
- *Models*
  - *Segmentation*
- *Registry*
- *Utilities*
  - *Argparse*
  - *Compression*
  - *Confusion Matrix*
  - *Dataset*
  - *Misc*
  - *Pandas*
  - *Parallel*
  - *Plotly*
  - *Serialization*
  - *Visualization*
- *Tools*
  - *Tensorboard*

4.8.1 DL

Callbacks

4.8.2 NN

Criterion

Modules

Optimizers

Schedulers

4.8.3 Models

Segmentation

4.8.4 Registry

catalyst subpackage registries

4.8.5 Utilities

Argparse

Compression

Confusion Matrix

Dataset

Misc

Pandas

Parallel

Plotly

Serialization

Visualization

4.8.6 Tools

Tensorboard

Python Module Index

C

catalyst.contrib.registry, 26
catalyst.data.augmentor, 19
catalyst.tools.frozen_class, 21
catalyst.tools.registry, 22
catalyst.tools.time_manager, 23

Index

Symbols

__init__() (*catalyst.data.augmentor.Augmentor method*), 19
__init__() (*catalyst.data.augmentor.AugmentorCompose method*), 19
__init__() (*catalyst.data.augmentor.AugmentorKeys method*), 19
__init__() (*catalyst.tools.registry.Registry method*), 22
__init__() (*catalyst.tools.registry.RegistryException method*), 23
__init__() (*catalyst.tools.time_manager.TimeManager method*), 23

A

add() (*catalyst.tools.registry.Registry method*), 22
add_from_module() (*catalyst.tools.registry.Registry method*), 22
all() (*catalyst.tools.registry.Registry method*), 22
Augmentor (*class in catalyst.data.augmentor*), 19
AugmentorCompose (*class in catalyst.data.augmentor*), 19
AugmentorKeys (*class in catalyst.data.augmentor*), 19

C

catalyst.contrib.registry (*module*), 26
catalyst.data.augmentor (*module*), 19
catalyst.tools.frozen_class (*module*), 21
catalyst.tools.registry (*module*), 22
catalyst.tools.time_manager (*module*), 23

F

FrozenClass (*class in catalyst.tools.frozen_class*), 21

G

get() (*catalyst.tools.registry.Registry method*), 22
get_from_params() (*catalyst.tools.registry.Registry method*), 22
get_if_str() (*catalyst.tools.registry.Registry method*), 23
get_instance() (*catalyst.tools.registry.Registry method*), 23

L

late_add() (*catalyst.tools.registry.Registry method*), 23
len() (*catalyst.tools.registry.Registry method*), 23

R

Registry (*class in catalyst.tools.registry*), 22
RegistryException, 23
reset() (*catalyst.tools.time_manager.TimeManager method*), 23

S

start() (*catalyst.tools.time_manager.TimeManager method*), 24
stop() (*catalyst.tools.time_manager.TimeManager method*), 24

T

TimeManager (*class in catalyst.tools.time_manager*), 23