inferno Documentation

Release 0.1.7

Nasim Rahaman

Jul 06, 2018

Contents

Contents:

CHAPTER 1

Inferno

Inferno is a little library providing utilities and convenience functions/classes around PyTorch. It's a work-in-progress, but the first stable release (0.2) is underway!

- Free software: Apache Software License 2.0
- Documentation: https://pytorch-inferno.readthedocs.io (Work in progress).

1.1 Features

Current features include:

- a basic Trainer class to encapsulate the training boilerplate (iteration/epoch loops, validation and checkpoint creation),
- a graph API for building models with complex architectures, powered by networkx.
- easy data-parallelism over multiple GPUs,
- a submodule for *torch.nn.Module*-level parameter initialization,
- a submodule for data preprocessing / transforms,
- support for Tensorboard (best with atleast tensorflow-cpu installed)
- a callback API to enable flexible interaction with the trainer,
- various utility layers with more underway,
- a submodule for volumetric datasets, and more!

```
import torch.nn as nn
from inferno.io.box.cifar import get_cifar10_loaders
from inferno.trainers.basic import Trainer
from inferno.trainers.callbacks.logging.tensorboard import TensorboardLogger
```

(continues on next page)

(continued from previous page)

```
from inferno.extensions.layers.convolutional import ConvELU2D
from inferno.extensions.layers.reshape import Flatten

# Fill these in:
LOG_DIRECTORY = '...'
SAVE_DIRECTORY = '...'
DATASET_DIRECTORY = '...'
DOWNLOAD_CIFAR = True
USE_CUDA = True

# Build torch model
model = nn.Sequential(
    ConvELU2D(in_channels=3, out_channels=256, kernel_size=3),
    nn.MaxPool2d(kernel_size=2, stride=2),
    ConvELU2D(in_channels=256, out_channels=256, kernel_size=3),
    nn.MaxPool2d(kernel_size=2, stride=2),
    ConvELU2D(in_channels=256, out_channels=256, kernel_size=3),
    nn.MaxPool2d(kernel_size=2, stride=2),
    Flatten(),
    nn.Linear(in_features=(256 * 4 * 4), out_features=10),
    nn.Softmax()
)

# Load loaders
train_loader, validate_loader = get_cifar10_loaders(DATASET_DIRECTORY,
                                                    download=DOWNLOAD_CIFAR)

# Build trainer
trainer = Trainer(model) \
  .build_criterion('CrossEntropyLoss') \
  .build_metric('CategoricalError') \
  .build_optimizer('Adam') \
  .validate_every((2, 'epochs')) \
  .save_every((5, 'epochs')) \
  .save_to_directory(SAVE_DIRECTORY) \
  .set_max_num_epochs(10) \
  .build_logger(TensorboardLogger(log_scalars_every=(1, 'iteration'),
                                  log_images_every='never'),
                log_directory=LOG_DIRECTORY)

# Bind loaders
trainer \
    .bind_loader('train', train_loader) \
    .bind_loader('validate', validate_loader)

if USE_CUDA:
  trainer.cuda()

# Go!
trainer.fit()
```

To visualize the training progress, navigate to *LOG_DIRECTORY* and fire up tensorboard with

```
$ tensorboard --logdir=${PWD} --port=6007
```

and navigate to *localhost:6007* with your browser.

1.2 Installation

Conda packages for linux and mac (only python 3) are available via

```
$ conda install -c inferno-pytorch inferno
```

1.3 Future Features:

Planned features include:

- a class to encapsulate Hogwild! training over multiple GPUs,
- minimal shape inference with a dry-run,
- proper packaging and documentation,
- cutting-edge fresh-off-the-press implementations of what the future has in store. :)

1.4 Credits

All contributors are listed here.

This packag was partially generated with Cookiecutter and the audreyr/cookiecutter-pypackage project template + lots of work by Thorsten.

CHAPTER 2

Installation

2.1 Install on Linux and OSX

2.1.1 Developers

First, make sure you have Pytorch installed.

Then, clone this repository with:

```
$ git clone https://github.com/nasimrahaman/inferno.git
```

Next, install the dependencies.

```
$ cd inferno
$ pip install -r requirements.txt
```

2.1.2 If you use python from the shell:

Finally, add *inferno* to your *PYTHONPATH* with:

```
source add2path.sh
```

2.1.3 If you use PyCharm:

Refer to this QA about setting up paths with Pycharm.

CHAPTER 3

Installation via PyPi / pip / setup.py(Experimental)

You need to install pytorch via pip before installing inferno. Follow the pytorch installation guide.

3.1 Stable release

To install inferno, run this command in your terminal:

```
$ pip install inferno-pytorch
```

This is the preferred method to install inferno, as it will always install the most recent stable release.

If you don't have pip installed, this Python installation guide can guide you through the process.

3.2 From sources

First, make sure you have Pytorch installed. The sources for inferno can be downloaded from the Github repo. You can either clone the public repository:

```
$ git clone git://github.com/nasimrahaman/inferno
```

Or download the tarball:

```
$ curl  -OL https://github.com/nasimrahaman/inferno/tarball/master
```

Once you have a copy of the source, you can install it with:

```
$ python setup.py install
```

CHAPTER 4

Usage

Inferno is a utility library built around [PyTorch](http://pytorch.org/), designed to help you train and even build complex pytorch models. And in this tutorial, we'll see how! If you're new to PyTorch, I highly recommended you work through the [Pytorch tutorials](http://pytorch.org/tutorials/) first.

4.1 Building a PyTorch Model

Inferno's training machinery works with just about any valid [Pytorch module](http://pytorch.org/docs/master/nn.html#torch.nn.Module). However, to make things even easier, we also provide pre-configured layers that work out-of-the-box. Let's use them to build a convolutional neural network for Cifar-10.

```
import torch.nn as nn
from inferno.extensions.layers.convolutional import ConvELU2D
from inferno.extensions.layers.reshape import Flatten
```

ConvELU2D is a 2-dimensional convolutional layer with orthogonal weight initialization and [ELU](http://pytorch.org/docs/master/nn.html#torch.nn.ELU) activation. *Flatten* reshapes the 4 dimensional activation tensor to a matrix. Let's use the Sequential container to chain together a bunch of convolutional and pooling layers, followed by a linear and softmax layer.

```
model = nn.Sequential(
    ConvELU2D(in_channels=3, out_channels=256, kernel_size=3),
    nn.MaxPool2d(kernel_size=2, stride=2),
    ConvELU2D(in_channels=256, out_channels=256, kernel_size=3),
    nn.MaxPool2d(kernel_size=2, stride=2),
    ConvELU2D(in_channels=256, out_channels=256, kernel_size=3),
    nn.MaxPool2d(kernel_size=2, stride=2),
    Flatten(),
    nn.Linear(in_features=(256 * 4 * 4), out_features=10),
    nn.Softmax()
)
```

Models this size don't win competitions anymore, but it'll do for our purpose.

4.1.1 Data Logistics

With our model built, it's time to worry about the data generators. Or is it?

```
from inferno.io.box.cifar import get_cifar10_loaders
train_loader, validate_loader = get_cifar10_loaders('path/to/cifar10',
                                                    download=True,
                                                    train_batch_size=128,
                                                    test_batch_size=100)
```

CIFAR-10 works out-of-the-*box* (pun very much intended) with all the fancy data-augmentation and normalization. Of course, it's perfectly fine if you have your own [*DataLoader*](http://pytorch.org/docs/master/data.html#torch.utils.data.DataLoader).

4.2 Preparing the Trainer

With our model and data loaders good to go, it's finally time to build the trainer. To start, let's initialize one.

```
from inferno.trainers.basic import Trainer

trainer = Trainer(model)
# Tell trainer about the data loaders
trainer.bind_loader('train', train_loader).bind_loader('validate', validate_loader)
```

Now to the things we could do with it.

4.2.1 Setting up Checkpointing

When training a model for days, it's usually a good idea to store the current training state to disk every once in a while. To set this up, we tell *trainer* where to store these *checkpoints* and how often.

```
trainer.save_to_directory('path/to/save/directory').save_every((25, 'epochs'))
```

So we're saving once every 25 epochs. But what if an epoch takes forever, and you don't wish to wait that long?

```
trainer.save_every((1000, 'iterations'))
```

In this setting, you're saving once every 1000 iterations (= batches). But we might also want to create a checkpoint when the validation score is the best. Easy as 1, 2,

```
trainer.save_at_best_validation_score()
```

Remember that a checkpoint contains the entire training state, and not just the model. Everything is included in the checkpoint file, including optimizer, criterion, and callbacks but __not the data **loaders**__.

4.2.2 Setting up Validation

Let's say you wish to validate once every 2 epochs.

```
trainer.validate_every((2, 'epochs'))
```

To be able to validate, you'll need to specify a validation metric.

```
trainer.build_metric('CategoricalError')
```

Inferno looks for a metric *'CategoricalError'* in *inferno.extensions.metrics*. To specify your own metric, subclass *inferno.extensions.metrics.base.Metric* and implement the *forward* method. With that done, you could:

```
trainer.build_metric(MyMetric)
```

or

```
trainer.build_metric(MyMetric, **my_metric_kwargs)
```

Note that the metric applies to 'torch.Tensor's, and not on 'torch.autograd.Variable's. Also, a metric might be way too expensive to evaluate every training iteration without slowing down the training. If this is the case and you'd like to evaluate the metric every (say) 10 *training* iterations:

```
trainer.evaluate_metric_every((10, 'iterations'))
```

However, while validating, the metric is evaluated once every iteration.

4.2.3 Setting up the Criterion and Optimizer

With that out of the way, let's set up a training criterion and an optimizer.

```
# set up the criterion
trainer.build_criterion('CrossEntropyLoss')
```

The *trainer* looks for a *'CrossEntropyLoss'* in *torch.nn*, which it finds. But any of the following would have worked:

```
trainer.build_criterion(nn.CrossEntropyLoss)
```

or

```
trainer.build_criterion(nn.CrossEntropyLoss())
```

What this means is that if you have your own loss criterion that has the same API as any of the criteria found in *torch.nn*, you should be fine by just plugging it in.

The same holds for the optimizer:

```
trainer.build_optimizer('Adam', weight_decay=0.0005)
```

Like for criteria, the *trainer* looks for a *'Adam'* in *torch.optim* (among other places), and initializes it with *model*'s parameters. Any keywords you might use for *torch.optim.Adam*, you could pass them to the *build_optimizer* method.

Or alternatively, you could use:

```
from torch.optim import Adam

trainer.build_optimizer(Adam, weight_decay=0.0005)
```

If you implemented your own optimizer (by subclassing *torch.optim.Optimizer*), you should be able to use it instead of *Adam*. Alternatively, if you already have an optimizer *instance*, you could do:

```
optimizer = MyOptimizer(model.parameters(), **optimizer_kwargs)
trainer.build_optimizer(optimizer)
```

4.2.4 Setting up Training Duration

You probably don't want to train forever, in which case you must specify:

```
trainer.set_max_num_epochs(100)
```

or

```
trainer.set_max_num_iterations(10000)
```

If you like to train indefinitely (or until you're happy with the results), use:

```
trainer.set_max_num_iterations('inf')
```

In this case, you'll need to interrupt the training manually with a *KeyboardInterrupt*.

4.2.5 Setting up Callbacks

Callbacks are pretty handy when it comes to interacting with the *Trainer*. More precisely: *Trainer* defines a number of events as 'triggers' for callbacks. Currently, these are:

```
BEGIN_OF_FIT,
END_OF_FIT,
BEGIN_OF_TRAINING_RUN,
END_OF_TRAINING_RUN,
BEGIN_OF_EPOCH,
END_OF_EPOCH,
BEGIN_OF_TRAINING_ITERATION,
END_OF_TRAINING_ITERATION,
BEGIN_OF_VALIDATION_RUN,
END_OF_VALIDATION_RUN,
BEGIN_OF_VALIDATION_ITERATION,
END_OF_VALIDATION_ITERATION,
BEGIN_OF_SAVE,
END_OF_SAVE
```

As an example, let's build a simple callback to interrupt the training on NaNs. We check at the end of every training iteration whether the training loss is NaN, and accordingly raise a *RuntimeError*.

```
import numpy as np
from inferno.trainers.callbacks.base import Callback

class NaNDetector(Callback):
    def end_of_training_iteration(self, **_):
        # The callback object has the trainer as an attribute.
        # The trainer populates its 'states' with torch tensors (NOT VARIABLES!)
        training_loss = self.trainer.get_state('training_loss')
        # Extract float from torch tensor
        training_loss = training_loss[0]
        if np.isnan(training_loss):
            raise RuntimeError("NaNs detected!")
```

With the callback defined, all we need to do is register it with the trainer:

```
trainer.register_callback(NaNDetector())
```

So the next time you get *RuntimeError: "NaNs detected!*, you know the drill.

4.2.6 Using Tensorboard

Inferno supports logging scalars and images to Tensorboard out-of-the-box, though this requires you have at least [tensorflow-cpu](https://github.com/tensorflow/tensorflow) installed. Let's say you want to log scalars every iteration and images every 20 iterations:

```
from inferno.trainers.callbacks.logging.tensorboard import TensorboardLogger

trainer.build_logger(TensorboardLogger(log_scalars_every=(1, 'iteration'),
                                       log_images_every=(20, 'iterations')),
                     log_directory='/path/to/log/directory')
```

After you've started training, use a bash shell to fire up tensorboard with:

```
$ tensorboard --logdir=/path/to/log/directory --port=6007
```

and navigate to *localhost:6007* with your favorite browser.

Fine print: missing the *log_images_every* keyword argument to *TensorboardLogger* will result in images being logged every iteration. If you don't have a fast hard drive, this might actually slow down the training. To not log images, just use *log_images_every='never'*.

4.2.7 Using GPUs

To use just one GPU:

```
trainer.cuda()
```

For multi-GPU data-parallel training, simply pass *trainer.cuda* a list of devices:

```
trainer.cuda(devices=[0, 1, 2, 3])
```

__Pro-**tip**__: Say you only want to use GPUs 0, 3, 5 and 7 (your colleagues might love you for this). Before running your training script, simply:

```
$ export CUDA_VISIBLE_DEVICES=0,3,5,7
$ python train.py
```

This maps device 0 to 0, 3 to 1, 5 to 2 and 7 to 3.

4.2.8 One more thing

Once you have everything configured, use

```
trainer.fit()
```

to commence training! This last step is kinda important. :wink:

4.3 Cherries:

4.3.1 Building Complex Models with the Graph API

Work in Progress:

4.3.2 Parameter Initialization

Work in Progress:

4.3.3 Support

Work in Progress:

CHAPTER 5

Inferno Examples Gallery

5.1 Gallery of Examples

Note: Click *here* to download the full example code

5.1.1 Trainer Example

This example should illustrate how to use the trainer class.

```
import torch.nn as nn
from inferno.io.box.cifar import get_cifar10_loaders
from inferno.trainers.basic import Trainer
from inferno.trainers.callbacks.logging.tensorboard import TensorboardLogger
from inferno.extensions.layers.convolutional import ConvELU2D
from inferno.extensions.layers.reshape import Flatten

# lil helper to make sure dirs exits
import os
def ensure_dir(file_path):
    directory = os.path.dirname(file_path)
    if not os.path.exists(directory):
        os.makedirs(directory)
    return directory

# change directories to your needs
LOG_DIRECTORY = ensure_dir('log')
SAVE_DIRECTORY = ensure_dir('save')
DATASET_DIRECTORY = ensure_dir('dataset')


DOWNLOAD_CIFAR = True
```

(continues on next page)

(continued from previous page)

```
USE_CUDA = True

# Build torch model
model = nn.Sequential(
    ConvELU2D(in_channels=3, out_channels=256, kernel_size=3),
    nn.MaxPool2d(kernel_size=2, stride=2),
    ConvELU2D(in_channels=256, out_channels=256, kernel_size=3),
    nn.MaxPool2d(kernel_size=2, stride=2),
    ConvELU2D(in_channels=256, out_channels=256, kernel_size=3),
    nn.MaxPool2d(kernel_size=2, stride=2),
    Flatten(),
    nn.Linear(in_features=(256 * 4 * 4), out_features=10),
    nn.Softmax()
)

# Load loaders
train_loader, validate_loader = get_cifar10_loaders(DATASET_DIRECTORY,
                                                    download=DOWNLOAD_CIFAR)

# Build trainer
trainer = Trainer(model) \
.build_criterion('CrossEntropyLoss') \
.build_metric('CategoricalError') \
.build_optimizer('Adam') \
.validate_every((2, 'epochs')) \
.save_every((5, 'epochs')) \
.save_to_directory(SAVE_DIRECTORY) \
.set_max_num_epochs(10) \
.build_logger(TensorboardLogger(log_scalars_every=(1, 'iteration'),
                                log_images_every='never'),
              log_directory=LOG_DIRECTORY)

# Bind loaders
trainer \
    .bind_loader('train', train_loader) \
    .bind_loader('validate', validate_loader)

if USE_CUDA:
    trainer.cuda()

# Go!
trainer.fit()
```

Total running time of the script: (0 minutes 0.000 seconds)

Note: Click *here* to download the full example code

5.1.2 Regularized MNIST Example

This example demonstrates adding and logging arbitrary regularization losses, in this case, L2 activity regularization and L1 weight regularization.

- Add a *_losses* dictionary to any module containing loss names and values
- Use a criterion from *inferno.extensions.criteria.regularized* that will collect and add those losses

- Call *Trainer.observe_training_and_validation_states* to log the losses as well

```
import argparse
import sys

import torch
import torch.nn as nn
from torchvision import datasets, transforms

from inferno.extensions.layers.reshape import Flatten
from inferno.trainers.basic import Trainer
from inferno.trainers.callbacks.logging.tensorboard import TensorboardLogger


class RegularizedLinear(nn.Linear):
    def __init__(self, *args, ar_weight=1e-3, l1_weight=1e-3, **kwargs):
        super(RegularizedLinear, self).__init__(*args, **kwargs)
        self.ar_weight = ar_weight
        self.l1_weight = l1_weight
        self._losses = {}

    def forward(self, input):
        output = super(RegularizedLinear, self).forward(input)
        self._losses['activity_regularization'] = (output * output).sum() * self.ar_
↪weight
        self._losses['l1_weight_regularization'] = torch.abs(self.weight).sum() *␣
↪self.l1_weight
        return output


def model_fn():
    return nn.Sequential(
        Flatten(),
        RegularizedLinear(in_features=784, out_features=256),
        nn.LeakyReLU(),
        RegularizedLinear(in_features=256, out_features=128),
        nn.LeakyReLU(),
        RegularizedLinear(in_features=128, out_features=10)
    )


def mnist_data_loaders(args):
    kwargs = {'num_workers': 1, 'pin_memory': True} if args.cuda else {}
    train_loader = torch.utils.data.DataLoader(
        datasets.MNIST('./data', train=True, download=True,
                       transform=transforms.Compose([
                           transforms.ToTensor(),
                           transforms.Normalize((0.1307,), (0.3081,))
                       ])),
        batch_size=args.batch_size, shuffle=True, **kwargs)
    test_loader = torch.utils.data.DataLoader(
        datasets.MNIST('./data', train=False, transform=transforms.Compose([
            transforms.ToTensor(),
            transforms.Normalize((0.1307,), (0.3081,))
        ])),
        batch_size=args.test_batch_size, shuffle=True, **kwargs)
    return train_loader, test_loader
```

(continues on next page)

(continued from previous page)

```
def train_model(args):
    model = model_fn()
    train_loader, validate_loader = mnist_data_loaders(args)

    # Build trainer
    trainer = Trainer(model) \
        .build_criterion('RegularizedCrossEntropyLoss') \
        .build_metric('CategoricalError') \
        .build_optimizer('Adam') \
        .validate_every((1, 'epochs')) \
        .save_every((1, 'epochs')) \
        .save_to_directory(args.save_directory) \
        .set_max_num_epochs(args.epochs) \
        .build_logger(TensorboardLogger(log_scalars_every=(1, 'iteration'),
                                        log_images_every='never'),
                      log_directory=args.save_directory)

    # Record regularization losses
    trainer.logger.observe_training_and_validation_states([
        'main_loss',
        'total_regularization_loss',
        'activity_regularization',
        'l1_weight_regularization'
    ])

    # Bind loaders
    trainer \
        .bind_loader('train', train_loader) \
        .bind_loader('validate', validate_loader)

    if args.cuda:
        trainer.cuda()

    # Go!
    trainer.fit()


def main(argv):
    # Training settings
    parser = argparse.ArgumentParser(description='PyTorch MNIST Example')
    parser.add_argument('--batch-size', type=int, default=64, metavar='N',
                        help='input batch size for training (default: 64)')
    parser.add_argument('--save-directory', type=str, default='output/mnist/v1',
                        help='output directory')
    parser.add_argument('--test-batch-size', type=int, default=1000, metavar='N',
                        help='input batch size for testing (default: 1000)')
    parser.add_argument('--epochs', type=int, default=20, metavar='N',
                        help='number of epochs to train (default: 20)')
    parser.add_argument('--no-cuda', action='store_true', default=False,
                        help='disables CUDA training')
    args = parser.parse_args(argv)
    args.cuda = not args.no_cuda and torch.cuda.is_available()
    train_model(args)


if __name__ == '__main__':
```

(continues on next page)

(continued from previous page)

```
    main(sys.argv[1:])
```

Total running time of the script: (0 minutes 0.000 seconds)

CHAPTER 6

Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

6.1 Types of Contributions

6.1.1 Report Bugs

Report bugs at https://github.com/nasimrahaman/inferno/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

6.1.2 Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" and "help wanted" is open to whoever wants to implement it.

6.1.3 Implement Features

Look through the GitHub issues for features. Anything tagged with "enhancement" and "help wanted" is open to whoever wants to implement it.

6.1.4 Write Documentation

inferno could always use more documentation, whether as part of the official inferno docs, in docstrings, or even on the web in blog posts, articles, and such.

6.1.5 Submit Feedback

The best way to send feedback is to file an issue at https://github.com/nasimrahaman/inferno/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

6.2 Get Started!

Ready to contribute? Here's how to set up *inferno* for local development.

1. Fork the *inferno* repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/inferno.git
```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

```
$ mkvirtualenv inferno
$ cd inferno/
$ python setup.py develop
```

4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions with tox:

```
$ flake8 inferno tests
$ python setup.py test or py.test
$ tox
```

   To get flake8 and tox, just pip install them into your virtualenv.

6. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website.

6.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 3.5 and 3.6. Check https://travis-ci.org/nasimrahaman/inferno/pull_requests and make sure that the tests pass for all supported Python versions.

6.4 Tips

To run a subset of tests:

```
$ python -m unittest tests.test_inferno
```

CHAPTER 7

inferno

7.1 inferno package

7.1.1 Subpackages

7.1.1.1 inferno.extensions package

Subpackages

inferno.extensions.containers package

Submodules

inferno.extensions.containers.graph module

class inferno.extensions.containers.graph.**NNGraph**(*incoming_graph_data=None*, ***attr*)

Bases: networkx.classes.digraph.DiGraph

A NetworkX DiGraph, except that node and edge ordering matters.

ATTRIBUTES_TO_NOT_COPY = {'payload'}

adjlist_dict_factory
alias of collections.OrderedDict

copy(***init_kwargs*)
Return a copy of the graph.

The copy method by default returns a shallow copy of the graph and attributes. That is, if an attribute is a container, that container is shared by the original an the copy. Use Python's *copy.deepcopy* for new containers.

If *as_view* is True then a view is returned instead of a copy.

Notes

All copies reproduce the graph structure, but data attributes may be handled in different ways. There are four types of copies of a graph that people might want.

Deepcopy – The default behavior is a "deepcopy" where the graph structure as well as all data attributes and any objects they might contain are copied. The entire graph object is new so that changes in the copy do not affect the original object. (see Python's copy.deepcopy)

Data Reference (Shallow) – For a shallow copy the graph structure is copied but the edge, node and graph attribute dicts are references to those in the original graph. This saves time and memory but could cause confusion if you change an attribute in one graph and it changes the attribute in the other. NetworkX does not provide this level of shallow copy.

Independent Shallow – This copy creates new independent attribute dicts and then does a shallow copy of the attributes. That is, any attributes that are containers are shared between the new graph and the original. This is exactly what *dict.copy()* provides. You can obtain this style copy using:

```
>>> G = nx.path_graph(5)
>>> H = G.copy()
>>> H = G.copy(as_view=False)
>>> H = nx.Graph(G)
>>> H = G.fresh_copy().__class__(G)
```

Fresh Data – For fresh data, the graph structure is copied while new empty data attribute dicts are created. The resulting graph is independent of the original and it has no edge, node or graph attributes. Fresh copies are not enabled. Instead use:

```
>>> H = G.fresh_copy()
>>> H.add_nodes_from(G)
>>> H.add_edges_from(G.edges)
```

View – Inspired by dict-views, graph-views act like read-only versions of the original graph, providing a copy of the original structure without requiring any memory for copying the information.

See the Python copy module for more information on shallow and deep copies, https://docs.python.org/2/library/copy.html.

Parameters **as_view** (*bool, optional (default=False)*) – If True, the returned graph-view provides a read-only view of the original graph without actually copying any data.

Returns **G** – A copy of the graph.

Return type *Graph*

See also:

to_directed() return a directed copy of the graph.

Examples

```
>>> G = nx.path_graph(4)  # or DiGraph, MultiGraph, MultiDiGraph, etc
>>> H = G.copy()
```

node_dict_factory
alias of collections.OrderedDict

class inferno.extensions.containers.graph.**Graph**(*graph=None*)
Bases: torch.nn.modules.module.Module

A graph structure to build networks with complex architectures. The resulting graph model can be used like any other *torch.nn.Module*. The graph structure used behind the scenes is a *networkx.DiGraph*. This internal graph is exposed by the *apply_on_graph* method, which can be used with any NetworkX function (e.g. for plotting with matplotlib or GraphViz).

Examples

The naive inception module (without the max-pooling for simplicity) with ELU-layers of 64 units can be built as following, (assuming 64 input channels):

```
>>> from inferno.extensions.layers.reshape import Concatenate
>>> from inferno.extensions.layers.convolutional import ConvELU2D
>>> import torch
>>> from torch.autograd import Variable
>>> # Build the model
>>> inception_module = Graph()
>>> inception_module.add_input_node('input')
>>> inception_module.add_node('conv1x1', ConvELU2D(64, 64, 3), previous='input')
>>> inception_module.add_node('conv3x3', ConvELU2D(64, 64, 3), previous='input')
>>> inception_module.add_node('conv5x5', ConvELU2D(64, 64, 3), previous='input')
>>> inception_module.add_node('cat', Concatenate(),
>>>                           previous=['conv1x1', 'conv3x3', 'conv5x5'])
>>> inception_module.add_output_node('output', 'cat')
>>> # Build dummy variable
>>> input = Variable(torch.rand(1, 64, 100, 100))
>>> # Get output
>>> output = inception_module(input)
```

add_edge(*from_node*, *to_node*)
Add an edge between two nodes.

Parameters

- **from_node** (*str*) – Name of the source node.
- **to_node** (*str*) – Name of the target node.

Returns self

Return type *Graph*

Raises AssertionError – if either of the two nodes is not in the graph, or if the edge is not 'legal'.

add_input_node(*name*)
Add an input to the graph. The order in which input nodes are added is the order in which the forward method accepts its inputs.

Parameters **name** (*str*) – Name of the input node.

Returns self

Return type *Graph*

add_node(*name*, *module*, *previous=None*)
Add a node to the graph.

Parameters

- **name** (*str*) – Name of the node. Nodes are identified by their names.
- **module** (*torch.nn.Module*) – Torch module for this node.
- **previous** (*str or list of str*) – (List of) name(s) of the previous node(s).

Returns self

Return type *Graph*

add_output_node(*name*, *previous=None*)
Add an output to the graph. The order in which output nodes are added is the order in which the forward method returns its outputs.

Parameters **name** (*str*) – Name of the output node.

Returns self

Return type *Graph*

apply_on_graph(*function*, **args*, ***kwargs*)
Applies a *function* on the internal graph.

assert_graph_is_valid()
Asserts that the graph is valid.

clear_payloads(*graph=None*)

forward(**inputs*)
Defines the computation performed at every call.

Should be overriden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

forward_through_node(*name*, *input=None*)

get_module_for_nodes(*names*)
Gets the *torch.nn.Module* object for nodes corresponding to *names*.

Parameters **names** (*str or list of str*) – Names of the nodes to fetch the modules of.

Returns Module or a list of modules corresponding to *names*.

Return type list or torch.nn.Module

get_parameters_for_nodes(*names*, *named=False*)
Get parameters of all nodes listed in *names*.

graph

graph_is_valid
Checks if the graph is valid.

input_nodes
Gets a list of input nodes. The order is relevant and is the same as that in which the forward method accepts its inputs.

Returns A list of names (str) of the input nodes.

Return type list

is_node_in_graph(*name*)
Checks whether a node is in the graph.

Parameters **name** (*str*) – Name of the node.

Returns

Return type bool

is_sink_node(*name*)
Checks whether a given node (by name) is a sink node. A sink node has no outgoing edges.

Parameters **name** (*str*) – Name of the node.

Returns

Return type bool

Raises AssertionError – if node is not found in the graph.

is_source_node(*name*)
Checks whether a given node (by name) is a source node. A source node has no incoming edges.

Parameters **name** (*str*) – Name of the node.

Returns

Return type bool

Raises AssertionError – if node is not found in the graph.

output_nodes
Gets a list of output nodes. The order is relevant and is the same as that in which the forward method returns its outputs.

Returns A list of names (str) of the output nodes.

Return type list

to_device(*names*, *target_device*, *device_ordinal=None*, *async=False*)
Transfer nodes in the network to a specified device.

inferno.extensions.containers.sequential module

class inferno.extensions.containers.sequential.**Sequential1**(**args*)
Bases: torch.nn.modules.container.Sequential

Like torch.nn.Sequential, but with a few extra methods.

class inferno.extensions.containers.sequential.**Sequential2**(**args*)
Bases: *inferno.extensions.containers.sequential.Sequential1*

Another sequential container. Identitcal to torch.nn.Sequential, except that modules may return multiple outputs and accept multiple inputs.

forward(**input*)
Defines the computation performed at every call.

Should be overriden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks

while the latter silently ignores them.

Module contents

inferno.extensions.criteria package

Submodules

inferno.extensions.criteria.core module

class inferno.extensions.criteria.core.**Criteria**(**criteria*)
Bases: torch.nn.modules.module.Module

Aggregate multiple criteria to one.

forward(*prediction*, *target*)
Defines the computation performed at every call.

Should be overriden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

class inferno.extensions.criteria.core.**As2DCriterion**(*criterion*)
Bases: torch.nn.modules.module.Module

Makes a given criterion applicable on (N, C, H, W) prediction and (N, H, W) target tensors, if they're applicable to (N, C) prediction and (N,) target tensors .

forward(*prediction*, *target*)
Defines the computation performed at every call.

Should be overriden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

inferno.extensions.criteria.set_similarity_measures module

class inferno.extensions.criteria.set_similarity_measures.**SorensenDiceLoss**(*weight=None*, *channelwise=True*, *eps=1e-06*)
Bases: torch.nn.modules.module.Module

Computes a loss scalar, which when minimized maximizes the Sorensen-Dice similarity between the input and the target. For both inputs and targets it must be the case that *input_or_target.size(1) = num_channels*.

forward(*input*, *target*)
Defines the computation performed at every call.

Should be overriden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

Module contents

inferno.extensions.initializers package

Submodules

inferno.extensions.initializers.base module

class inferno.extensions.initializers.base.**Initializer**
Bases: `object`

Base class for all initializers.

VALID_LAYERS = {'ConvTranspose2d', 'Linear', 'Embedding', 'ConvTranspose1d', 'Conv2d',

call_on_bias(*tensor*)

call_on_tensor(*tensor*)

call_on_weight(*tensor*)

classmethod initializes_bias()

classmethod initializes_weight()

class inferno.extensions.initializers.base.**Initialization**(*weight_initializer=None*, *bias_initializer=None*)
Bases: *inferno.extensions.initializers.base.Initializer*

call_on_bias(*tensor*)

call_on_weight(*tensor*)

class inferno.extensions.initializers.base.**WeightInitFunction**(*init_function*, **init_function_args*, ***init_function_kwargs*)
Bases: *inferno.extensions.initializers.base.Initializer*

call_on_weight(*tensor*)

class inferno.extensions.initializers.base.**BiasInitFunction**(*init_function*, **init_function_args*, ***init_function_kwargs*)
Bases: *inferno.extensions.initializers.base.Initializer*

call_on_bias(*tensor*)

class inferno.extensions.initializers.base.**TensorInitFunction**(*init_function*, **init_function_args*, ***init_function_kwargs*)

Bases: *inferno.extensions.initializers.base.Initializer*

call_on_tensor(*tensor*)

inferno.extensions.initializers.presets module

class inferno.extensions.initializers.presets.**Constant**(*constant*)

Bases: *inferno.extensions.initializers.base.Initializer*

Initialize with a constant.

call_on_tensor(*tensor*)

class inferno.extensions.initializers.presets.**NormalWeights**(*mean=0.0*, *stddev=1.0*, *sqrt_gain_over_fan_in=None*)

Bases: *inferno.extensions.initializers.base.Initializer*

Initialize weights with random numbers drawn from the normal distribution at *mean* and *stddev*.

call_on_weight(*tensor*)

compute_fan_in(*tensor*)

class inferno.extensions.initializers.presets.**SELUWeightsZeroBias**

Bases: *inferno.extensions.initializers.base.Initialization*

class inferno.extensions.initializers.presets.**ELUWeightsZeroBias**

Bases: *inferno.extensions.initializers.base.Initialization*

class inferno.extensions.initializers.presets.**OrthogonalWeightsZeroBias**(*orthogonal_gain=1.0*)

Bases: *inferno.extensions.initializers.base.Initialization*

class inferno.extensions.initializers.presets.**KaimingNormalWeightsZeroBias**(*relu_leakage=0*)

Bases: *inferno.extensions.initializers.base.Initialization*

Module contents

inferno.extensions.layers package

Submodules

inferno.extensions.layers.activations module

class inferno.extensions.layers.activations.**SELU**

Bases: torch.nn.modules.module.Module

forward(*input*)

Defines the computation performed at every call.

Should be overriden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

static selu(*x*)

inferno.extensions.layers.convolutional module

class inferno.extensions.layers.convolutional.**ConvActivation**(*in_channels*, *out_channels*, *kernel_size*, *dim*, *activation*, *stride=1*, *dilation=1*, *groups=None*, *depthwise=False*, *bias=True*, *deconv=False*, *initialization=None*)

Bases: `torch.nn.modules.module.Module`

Convolutional layer with 'SAME' padding followed by an activation.

forward(*input*)

Defines the computation performed at every call.

Should be overriden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

get_padding(*kernel_size*, *dilation*)

class inferno.extensions.layers.convolutional.**ConvELU2D**(*in_channels*, *out_channels*, *kernel_size*)

Bases: *inferno.extensions.layers.convolutional.ConvActivation*

2D Convolutional layer with 'SAME' padding, ELU and orthogonal weight initialization.

class inferno.extensions.layers.convolutional.**ConvELU3D**(*in_channels*, *out_channels*, *kernel_size*)

Bases: *inferno.extensions.layers.convolutional.ConvActivation*

3D Convolutional layer with 'SAME' padding, ELU and orthogonal weight initialization.

class inferno.extensions.layers.convolutional.**ConvSigmoid2D**(*in_channels*, *out_channels*, *kernel_size*)

Bases: *inferno.extensions.layers.convolutional.ConvActivation*

2D Convolutional layer with 'SAME' padding, Sigmoid and orthogonal weight initialization.

class inferno.extensions.layers.convolutional.**ConvSigmoid3D**(*in_channels*, *out_channels*, *kernel_size*)

Bases: *inferno.extensions.layers.convolutional.ConvActivation*

3D Convolutional layer with 'SAME' padding, Sigmoid and orthogonal weight initialization.

class inferno.extensions.layers.convolutional.**DeconvELU2D**(*in_channels*, *out_channels*, *kernel_size=2*)

Bases: *inferno.extensions.layers.convolutional.ConvActivation*

2D deconvolutional layer with ELU and orthogonal weight initialization.

class inferno.extensions.layers.convolutional.**DeconvELU3D**(*in_channels*, *out_channels*, *kernel_size=2*)

Bases: *inferno.extensions.layers.convolutional.ConvActivation*

3D deconvolutional layer with ELU and orthogonal weight initialization.

class inferno.extensions.layers.convolutional.**StridedConvELU2D**(*in_channels*, *out_channels*, *kernel_size*, *stride=2*)

Bases: *inferno.extensions.layers.convolutional.ConvActivation*

2D strided convolutional layer with 'SAME' padding, ELU and orthogonal weight initialization.

class inferno.extensions.layers.convolutional.**StridedConvELU3D**(*in_channels*, *out_channels*, *kernel_size*, *stride=2*)

Bases: *inferno.extensions.layers.convolutional.ConvActivation*

2D strided convolutional layer with 'SAME' padding, ELU and orthogonal weight initialization.

class inferno.extensions.layers.convolutional.**DilatedConvELU2D**(*in_channels*, *out_channels*, *kernel_size*, *dilation=2*)

Bases: *inferno.extensions.layers.convolutional.ConvActivation*

2D dilated convolutional layer with 'SAME' padding, ELU and orthogonal weight initialization.

class inferno.extensions.layers.convolutional.**DilatedConvELU3D**(*in_channels*, *out_channels*, *kernel_size*, *dilation=2*)

Bases: *inferno.extensions.layers.convolutional.ConvActivation*

3D dilated convolutional layer with 'SAME' padding, ELU and orthogonal weight initialization.

class inferno.extensions.layers.convolutional.**Conv2D**(*in_channels*, *out_channels*, *kernel_size*, *dilation=1*, *activation=None*)

Bases: *inferno.extensions.layers.convolutional.ConvActivation*

2D convolutional layer with same padding and orthogonal weight initialization. By default, this layer does not apply an activation function.

class inferno.extensions.layers.convolutional.**Conv3D**(*in_channels*, *out_channels*, *kernel_size*, *dilation=1*, *activation=None*)

Bases: *inferno.extensions.layers.convolutional.ConvActivation*

3D convolutional layer with same padding and orthogonal weight initialization. By default, this layer does not apply an activation function.

class inferno.extensions.layers.convolutional.**BNReLUConv2D**(*in_channels*, *out_channels*, *kernel_size*)

Bases: *inferno.extensions.layers.convolutional.ConvActivation*

2D BN-ReLU-Conv layer with 'SAME' padding and He weight initialization.

forward(*input*)

Defines the computation performed at every call.

Should be overriden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

class inferno.extensions.layers.convolutional.**BNReLUConv3D**(*in_channels*, *out_channels*, *kernel_size*)

Bases: *inferno.extensions.layers.convolutional.ConvActivation*

3D BN-ReLU-Conv layer with 'SAME' padding and He weight initialization.

forward(*input*)

Defines the computation performed at every call.

Should be overriden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

class inferno.extensions.layers.convolutional.**BNReLUDepthwiseConv2D**(*in_channels*, *out_channels*, *kernel_size*)

Bases: *inferno.extensions.layers.convolutional.ConvActivation*

2D BN-ReLU-Conv layer with 'SAME' padding, He weight initialization and depthwise convolution. Note that depthwise convolutions require *in_channels* == *out_channels*.

forward(*input*)

Defines the computation performed at every call.

Should be overriden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

class inferno.extensions.layers.convolutional.**ConvSELU2D**(*in_channels*, *out_channels*, *kernel_size*)

Bases: *inferno.extensions.layers.convolutional.ConvActivation*

2D Convolutional layer with SELU activation and the appropriate weight initialization.

class inferno.extensions.layers.convolutional.**ConvSELU3D**(*in_channels*, *out_channels*, *kernel_size*)

Bases: *inferno.extensions.layers.convolutional.ConvActivation*

3D Convolutional layer with SELU activation and the appropriate weight initialization.

inferno.extensions.layers.device module

class inferno.extensions.layers.device.**DeviceTransfer**(*target_device*, *device_ordinal=None*, *async=False*)

Bases: torch.nn.modules.module.Module

Layer to transfer variables to a specified device.

forward(**inputs*)

Defines the computation performed at every call.

Should be overriden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

class inferno.extensions.layers.device.**OnDevice**(*module*, *target_device*, *device_ordinal=None*, *async=False*)

Bases: torch.nn.modules.module.Module

Moves a module to a device. The advantage of using this over *torch.nn.Module.cuda* is that the inputs are transferred to the same device as the module, enabling easy model parallelism.

forward(**inputs*)

Defines the computation performed at every call.

Should be overriden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

transfer_module(*module*)

inferno.extensions.layers.reshape module

class inferno.extensions.layers.reshape.**View**(*as_shape*)

Bases: torch.nn.modules.module.Module

forward(*input*)

Defines the computation performed at every call.

Should be overriden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

validate_as_shape(*as_shape*)

class inferno.extensions.layers.reshape.**AsMatrix**
Bases: *inferno.extensions.layers.reshape.View*

class inferno.extensions.layers.reshape.**Flatten**
Bases: *inferno.extensions.layers.reshape.View*

class inferno.extensions.layers.reshape.**As3D**(*channel_as_z=False*, *num_channels_or_num_z_slices=1*)
Bases: torch.nn.modules.module.Module

forward(*input*)
Defines the computation performed at every call.

Should be overriden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

class inferno.extensions.layers.reshape.**As2D**(*z_as_channel=True*)
Bases: torch.nn.modules.module.Module

forward(*input*)
Defines the computation performed at every call.

Should be overriden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

class inferno.extensions.layers.reshape.**Concatenate**(*dim=1*)
Bases: torch.nn.modules.module.Module

Concatenate input tensors along a specified dimension.

forward(**inputs*)
Defines the computation performed at every call.

Should be overriden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

class inferno.extensions.layers.reshape.**Cat**(*dim=1*)
Bases: *inferno.extensions.layers.reshape.Concatenate*

An alias for *Concatenate*. Hey, everyone knows who Cat is.

class inferno.extensions.layers.reshape.**ResizeAndConcatenate**(*target_size*, *pool_mode='average'*)

Bases: torch.nn.modules.module.Module

Resize input tensors spatially (to a specified target size) before concatenating them along the channel dimension. The downsampling mode can be specified ('average' or 'max'), but the upsampling is always 'nearest'.

POOL_MODE_MAPPING = {'average': 'avg', 'avg': 'avg', 'max': 'max', 'mean': 'avg'}

forward(**inputs*)

Defines the computation performed at every call.

Should be overriden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

class inferno.extensions.layers.reshape.**PoolCat**(*target_size*, *pool_mode='average'*)

Bases: *inferno.extensions.layers.reshape.ResizeAndConcatenate*

Alias for *ResizeAndConcatenate*, just to annoy snarky web developers.

class inferno.extensions.layers.reshape.**Sum**

Bases: torch.nn.modules.module.Module

Sum all inputs.

forward(**inputs*)

Defines the computation performed at every call.

Should be overriden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

class inferno.extensions.layers.reshape.**SplitChannels**(*channel_index*)

Bases: torch.nn.modules.module.Module

Split input at a given index along the channel axis.

forward(*input*)

Defines the computation performed at every call.

Should be overriden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

Module contents

inferno.extensions.metrics package

Submodules

inferno.extensions.metrics.arand module

class inferno.extensions.metrics.arand.**ArandError**
Bases: *inferno.extensions.metrics.arand.ArandScore*

Arand Error = 1 - <arand score>

forward(*prediction*, *target*)

class inferno.extensions.metrics.arand.**ArandScore**
Bases: *inferno.extensions.metrics.base.Metric*

Arand Score, as defined in [1].

References

[1]: http://journal.frontiersin.org/article/10.3389/fnana.2015.00142/full#h3

forward(*prediction*, *target*)

inferno.extensions.metrics.arand.**adapted_rand**(*seg*, *gt*)

Compute Adapted Rand error as defined by the SNEMI3D contest [1] Formula is given as 1 - the maximal F-score of the Rand index (excluding the zero component of the original labels). Adapted from the SNEMI3D MATLAB script, hence the strange style.

seg [np.ndarray] the segmentation to score, where each value is the label at that point

gt [np.ndarray, same shape as seg] the groundtruth to score against, where each value is a label

are [float] The adapted Rand error; equal to $1 -

rac{2pr}{p + r}$,

where p and r are the precision and recall described below.

prec [float, optional] The adapted Rand precision.

rec [float, optional] The adapted Rand recall.

[1]: http://brainiac2.mit.edu/SNEMI3D/evaluation

inferno.extensions.metrics.base module

class inferno.extensions.metrics.base.**Metric**
Bases: object

forward(**args*, ***kwargs*)

inferno.extensions.metrics.categorical module

class inferno.extensions.metrics.categorical.**CategoricalError**(*aggregation_mode='mean'*)
Bases: *inferno.extensions.metrics.base.Metric*

Categorical error.

forward(*prediction*, *target*)

class inferno.extensions.metrics.categorical.**IOU**(*ignore_class=None*, *sharpen_prediction=False*, *eps=1e-06*)
Bases: *inferno.extensions.metrics.base.Metric*

Intersection over Union.

forward(*prediction*, *target*)

class inferno.extensions.metrics.categorical.**NegativeIOU**(*ignore_class=None*, *sharpen_prediction=False*, *eps=1e-06*)
Bases: *inferno.extensions.metrics.categorical.IOU*

forward(*prediction*, *target*)

Module contents

inferno.extensions.optimizers package

Submodules

inferno.extensions.optimizers.adam module

class inferno.extensions.optimizers.adam.**Adam**(*params*, *lr=0.001*, *betas=(0.9, 0.999)*, *eps=1e-08*, *lambda_l1=0*, *weight_decay=0*, ***kwargs*)
Bases: `torch.optim.optimizer.Optimizer`

Implements Adam algorithm with the option of adding a L1 penalty.

It has been proposed in Adam: A Method for Stochastic Optimization.

Parameters

- **params** (*iterable*) – iterable of parameters to optimize or dicts defining parameter groups
- **lr** (*float, optional*) – learning rate (default: 1e-3)
- **betas** (*Tuple[float, float], optional*) – coefficients used for computing running averages of gradient and its square (default: (0.9, 0.999))
- **eps** (*float, optional*) – term added to the denominator to improve numerical stability (default: 1e-8)
- **weight_decay** (*float, optional*) – weight decay (L2 penalty) (default: 0)

step(*closure=None*)
Performs a single optimization step.

Parameters **closure** (*callable, optional*) – A closure that reevaluates the model and returns the loss.

inferno.extensions.optimizers.annealed_adam module

class inferno.extensions.optimizers.annealed_adam.**AnnealedAdam**(*params*, *lr=0.001*, *betas=(0.9, 0.999)*, *eps=1e-08*, *lambda_l1=0*, *weight_decay=0*, *lr_decay=1.0*)

Bases: *inferno.extensions.optimizers.adam.Adam*

Implements Adam algorithm with learning rate annealing and optional L1 penalty.

It has been proposed in Adam: A Method for Stochastic Optimization.

Parameters

- **params** (*iterable*) – iterable of parameters to optimize or dicts defining parameter groups
- **lr** (*float, optional*) – learning rate (default: 1e-3)
- **betas** (*Tuple[float, float], optional*) – coefficients used for computing running averages of gradient and its square (default: (0.9, 0.999))
- **eps** (*float, optional*) – term added to the denominator to improve numerical stability (default: 1e-8)
- **lambda_l1** (*float, optional*) – L1 penalty (default: 0)
- **weight_decay** (*float, optional*) – L2 penalty (weight decay) (default: 0)
- **lr_decay** (*float, optional*) – decay learning rate by this factor after every step (default: 1.)

step(*closure=None*)

Performs a single optimization step.

Parameters **closure** (*callable, optional*) – A closure that reevaluates the model and returns the loss.

Module contents

Module contents

7.1.1.2 inferno.io package

Subpackages

inferno.io.box package

Submodules

inferno.io.box.camvid module

class inferno.io.box.camvid.**CamVid**(*root*, *split='train'*, *image_transform=None*, *label_transform=None*, *joint_transform=None*, *download=False*, *loader=<function default_loader>*)

Bases: torch.utils.data.dataset.Dataset

CLASSES = ['Sky', 'Building', 'Column-Pole', 'Road', 'Sidewalk', 'Tree', 'Sign-Symbol'

CLASS_WEIGHTS = [0.58872014284134, 0.51052379608154, 2.6966278553009, 0.45021694898605

MEAN = [0.41189489566336, 0.4251328133025, 0.4326707089857]

SPLIT_NAME_MAPPING = {'test': 'test', 'testing': 'test', 'train': 'train', 'trainin

STD = [0.27413549931506, 0.28506257482912, 0.28284674400252]

download()

inferno.io.box.camvid.**get_camvid_loaders**(*root_directory*, *image_shape=(360, 480)*, *labels_as_onehot=False*, *train_batch_size=1*, *validate_batch_size=1*, *test_batch_size=1*, *num_workers=2*)

inferno.io.box.camvid.**label_to_long_tensor**(*pic*)

inferno.io.box.camvid.**label_to_pil_image**(*label*)

inferno.io.box.camvid.**make_dataset**(*dir*)

inferno.io.box.cifar module

inferno.io.box.cifar.**get_cifar100_loaders**(*root_directory*, *train_batch_size=128*, *test_batch_size=100*, *download=False*, *augment=False*, *validation_dataset_size=None*)

inferno.io.box.cifar.**get_cifar10_loaders**(*root_directory*, *train_batch_size=128*, *test_batch_size=256*, *download=False*, *augment=False*, *validation_dataset_size=None*)

inferno.io.box.cityscapes module

class inferno.io.box.cityscapes.**Cityscapes**(*root_folder*, *split='train'*, *read_from_zip_archive=True*, *image_transform=None*, *label_transform=None*, *joint_transform=None*)

Bases: torch.utils.data.dataset.Dataset

BLACKLIST = ['leftImg8bit/train_extra/troisdorf/troisdorf_000000_000073_leftImg8bit.pn

CLASSES = {-1: 'license plate', 0: 'unlabeled', 1: 'ego vehicle', 2: 'rectificatio

MEAN = [0.28689554, 0.32513303, 0.28389177]

SPLIT_NAME_MAPPING = {'test': 'test', 'testing': 'test', 'train': 'train', 'train_e

STD = [0.18696375, 0.19017339, 0.18720214]

download()

get_image_and_label_roots()

inferno.io.box.cityscapes.**extract_image**(*path*, *image_path*)

inferno.io.box.cityscapes.**get_cityscapes_loaders**(*root_directory*, *image_shape=(1024, 2048)*, *labels_as_onehot=False*, *include_coarse_dataset=False*, *read_from_zip_archive=True*, *train_batch_size=1*, *validate_batch_size=1*, *num_workers=2*)

inferno.io.box.cityscapes.**get_filelist**(*path*)

inferno.io.box.cityscapes.**get_matching_labelimage_file**(*f*, *groundtruth*)

inferno.io.box.cityscapes.**make_dataset**(*path*, *split*)

inferno.io.box.cityscapes.**make_transforms**(*image_shape*, *labels_as_onehot*)

Module contents

Things that work out of the box. ;)

inferno.io.core package

Submodules

inferno.io.core.base module

class inferno.io.core.base.**IndexSpec**(*index=None*, *base_sequence_at_index=None*)

Bases: object

Class to wrap any extra index information a *Dataset* object might want to send back. This could be useful in (say) inference, where we would wish to (asynchronously) know more about the current input.

class inferno.io.core.base.**SyncableDataset**

Bases: torch.utils.data.dataset.Dataset

sync_with(*dataset*)

inferno.io.core.concatenate module

class inferno.io.core.concatenate.**Concatenate**(**datasets*, *transforms=None*)
Bases: torch.utils.data.dataset.Dataset

Concatenates mutliple datasets to one. This class does not implement synchronization primitives.

map_index(*index*)

inferno.io.core.data_utils module

inferno.io.core.data_utils.**defines_base_sequence**(*dataset*)

inferno.io.core.data_utils.**implements_sync_primitives**(*dataset*)

inferno.io.core.zip module

class inferno.io.core.zip.**Zip**(**datasets*, *sync=False*, *transforms=None*)
Bases: *inferno.io.core.base.SyncableDataset*

Zip two or more datasets to one dataset. If the datasets implement synchronization primitives, they are all synchronized with the first dataset.

sync_datasets()

sync_with(*dataset*)

class inferno.io.core.zip.**ZipReject**(**datasets*, *sync=False*, *transforms=None*, *rejection_dataset_indices*, *rejection_criterion*)
Bases: *inferno.io.core.zip.Zip*

Extends *Zip* by the functionality of rejecting samples that don't fulfill a specified rejection criterion.

fetch_from_rejection_datasets(*index*)

Module contents

inferno.io.transform package

Submodules

inferno.io.transform.base module

class inferno.io.transform.base.**Compose**(**transforms*)
Bases: object

Composes multiple callables (including but not limited to *Transform* objects).

add(*transform*)

remove(*name*)

class inferno.io.transform.base.**DTypeMapping**
Bases: object

DTYPE_MAPPING = {'byte': 'uint8', 'double': 'float64', 'float': 'float32', 'float16

class inferno.io.transform.base.**Transform**(*apply_to=None*)
Bases: object

Base class for a Transform. The argument *apply_to* (list) specifies the indices of the tensors this transform will be applied to.

The following methods are recognized (in order of descending priority):

- *batch_function*: Applies to all tensors in a batch simultaneously
- *tensor_function*: Applies to just __one__ tensor at a time.
- *volume_function*: For 3D volumes, applies to just __one__ volume at a time.
- *image_function*: For 2D or 3D volumes, applies to just __one__ image at a time.

For example, if both *volume_function* and *image_function* are defined, this means that only the former will be called. If the inputs are therefore not 5D batch-tensors of 3D volumes, a *NotImplementedError* is raised.

build_random_variables(***kwargs*)

clear_random_variables()

get_random_variable(*key*, *default=None*, *build=True*, ***random_variable_building_kwargs*)

set_random_variable(*key*, *value*)

inferno.io.transform.generic module

class inferno.io.transform.generic.**AsTorchBatch**(*dimensionality*, *add_channel_axis_if_necessary=True*, ***super_kwargs*)
Bases: *inferno.io.transform.base.Transform*

Converts a given numpy array to a torch batch tensor.

The result is a torch tensor __without__ the leading batch axis. For example, if the input is an image of shape *(100, 100)*, the output is a batch of shape *(1, 100, 100)*. The collate function will add the leading batch axis to obtain a tensor of shape *(N, 1, 100, 100)*, where *N* is the batch-size.

tensor_function(*tensor*)

class inferno.io.transform.generic.**Cast**(*dtype='float'*, ***super_kwargs*)
Bases: *inferno.io.transform.base.Transform*, *inferno.io.transform.base.DTypeMapping*

Casts inputs to a specified datatype.

tensor_function(*tensor*)

class inferno.io.transform.generic.**Label2OneHot**(*num_classes*, *dtype='float'*, ***super_kwargs*)
Bases: *inferno.io.transform.base.Transform*, *inferno.io.transform.base.DTypeMapping*

Convert integer labels to one-hot vectors for arbitrary dimensional data.

tensor_function(*tensor*)

class inferno.io.transform.generic.**Normalize**(*eps=0.0001*, *mean=None*, *std=None*, ***super_kwargs*)
Bases: *inferno.io.transform.base.Transform*

Normalizes input to zero mean unit variance.

tensor_function(*tensor*)

class inferno.io.transform.generic.**NormalizeRange**(*normalize_by=255.0*, ***super_kwargs*)

Bases: *inferno.io.transform.base.Transform*

Normalizes input by a constant.

tensor_function(*tensor*)

class inferno.io.transform.generic.**Project**(*projection*, ***super_kwargs*)

Bases: *inferno.io.transform.base.Transform*

Given a projection mapping (i.e. a dict) and an input tensor, this transform replaces all values in the tensor that equal a key in the mapping with the value corresponding to the key.

tensor_function(*tensor*)

inferno.io.transform.image module

class inferno.io.transform.image.**AdditiveGaussianNoise**(*sigma*, ***super_kwargs*)

Bases: *inferno.io.transform.base.Transform*

Add gaussian noise to the input.

build_random_variables(***kwargs*)

image_function(*image*)

class inferno.io.transform.image.**BinaryDilation**(*num_iterations=1*, *morphology_kwargs=None*, ***super_kwargs*)

Bases: *inferno.io.transform.image.BinaryMorphology*

Apply a binary dilation operation on an image.

class inferno.io.transform.image.**BinaryErosion**(*num_iterations=1*, *morphology_kwargs=None*, ***super_kwargs*)

Bases: *inferno.io.transform.image.BinaryMorphology*

Apply a binary erosion operation on an image.

class inferno.io.transform.image.**BinaryMorphology**(*mode*, *num_iterations=1*, *morphology_kwargs=None*, ***super_kwargs*)

Bases: *inferno.io.transform.base.Transform*

Apply a binary morphology operation on an image. Supported operations are dilation and erosion.

image_function(*image*)

class inferno.io.transform.image.**CenterCrop**(*size*, ***super_kwargs*)

Bases: *inferno.io.transform.base.Transform*

Crop patch of size *size* from the center of the image

image_function(*image*)

class inferno.io.transform.image.**ElasticTransform**(*alpha*, *sigma*, *order=1*, *invert=False*, ***super_kwargs*)

Bases: *inferno.io.transform.base.Transform*

Random Elastic Transformation.

NATIVE_DTYPES = {'float32', 'float64'}

PREFERRED_DTYPE = 'float32'

build_random_variables(***kwargs*)

cast(*image*)

image_function(*image*)

uncast(*image*)

class inferno.io.transform.image.**PILImage2NumPyArray**(*apply_to=None*)
Bases: *inferno.io.transform.base.Transform*

Convert a PIL Image object to a numpy array.

For images with multiple channels (say RGB), the channel axis is moved to front. Therefore, a (100, 100, 3) RGB image becomes an array of shape (3, 100, 100).

tensor_function(*tensor*)

class inferno.io.transform.image.**RandomCrop**(*output_image_shape*, ***super_kwargs*)
Bases: *inferno.io.transform.base.Transform*

Crop input to a given size.

This is similar to torchvision.transforms.RandomCrop, except that it operates on numpy arrays instead of PIL images. If you do have a PIL image and wish to use this transform, consider applying *PILImage2NumPyArray* first.

> **Warning:** If *output_image_shape* is larger than the image itself, the image is not cropped (along the relevant dimensions).

build_random_variables(*height_leeway*, *width_leeway*)

clear_random_variables()

image_function(*image*)

class inferno.io.transform.image.**RandomFlip**(*allow_lr_flips=True*, *allow_ud_flips=True*, ***super_kwargs*)
Bases: *inferno.io.transform.base.Transform*

Random left-right or up-down flips.

build_random_variables(***kwargs*)

image_function(*image*)

class inferno.io.transform.image.**RandomGammaCorrection**(*gamma_between=(0.5, 2.0)*, *gain=1*, ***super_kwargs*)
Bases: *inferno.io.transform.base.Transform*

Applies gamma correction [1] with a random gamma.

This transform uses *skimage.exposure.adjust_gamma*, which requires the input be positive.

References

[1] https://en.wikipedia.org/wiki/Gamma_correction

build_random_variables()

image_function(*image*)

class inferno.io.transform.image.**RandomRotate**(***super_kwargs*)
Bases: *inferno.io.transform.base.Transform*

Random 90-degree rotations.

build_random_variables(***kwargs*)

image_function(*image*)

class inferno.io.transform.image.**RandomSizedCrop**(*ratio_between=None*, *height_ratio_between=None*, *width_ratio_between=None*, *preserve_aspect_ratio=False*, *relative_target_aspect_ratio=None*, ***super_kwargs*)
Bases: *inferno.io.transform.base.Transform*

Extract a randomly sized crop from the image.

The ratio of the sizes of the cropped and the original image can be limited within specified bounds along both axes. To resize back to a constant sized image, compose with *Scale*.

build_random_variables(*image_shape*)

image_function(*image*)

class inferno.io.transform.image.**RandomTranspose**(***super_kwargs*)
Bases: *inferno.io.transform.base.Transform*

Random 2d transpose.

build_random_variables(***kwargs*)

image_function(*image*)

class inferno.io.transform.image.**Scale**(*output_image_shape*, *interpolation_order=3*, *zoom_kwargs=None*, ***super_kwargs*)
Bases: *inferno.io.transform.base.Transform*

Scales an image to a given size with spline interpolation of requested order.

Unlike torchvision.transforms.Scale, this does not depend on PIL and therefore works with numpy arrays. If you do have a PIL image and wish to use this transform, consider applying *PILImage2NumPyArray* first.

> **Warning:** This transform uses *scipy.ndimage.zoom* and requires scipy >= 0.13.0 to work correctly.

image_function(*image*)

inferno.io.transform.volume module

class inferno.io.transform.volume.**CentralSlice**(*apply_to=None*)
Bases: *inferno.io.transform.base.Transform*

volume_function(*volume*)

class inferno.io.transform.volume.**RandomFlip3D**(***super_kwargs*)
Bases: *inferno.io.transform.base.Transform*

build_random_variables(***kwargs*)

volume_function(*volume*)

class inferno.io.transform.volume.**VolumeAsymmetricCrop**(*crop_left*, *crop_right*, ***super_kwargs*)

Bases: *inferno.io.transform.base.Transform*

Crop *crop_left* from the left borders and *crop_right* from the right borders

volume_function(*volume*)

class inferno.io.transform.volume.**VolumeCenterCrop**(*size*, ***super_kwargs*)

Bases: *inferno.io.transform.base.Transform*

Crop patch of size *size* from the center of the volume

volume_function(*volume*)

Module contents

inferno.io.volumetric package

Submodules

inferno.io.volumetric.volume module

class inferno.io.volumetric.volume.**HDF5VolumeLoader**(*path*, *path_in_h5_dataset=None*, *data_slice=None*, *transforms=None*, *name=None*, ***slicing_config*)

Bases: *inferno.io.volumetric.volume.VolumeLoader*

class inferno.io.volumetric.volume.**TIFVolumeLoader**(*path*, *data_slice=None*, *transforms=None*, *name=None*, ***slicing_config*)

Bases: *inferno.io.volumetric.volume.VolumeLoader*

Loader for volumes stored in .tif files.

class inferno.io.volumetric.volume.**VolumeLoader**(*volume*, *window_size*, *stride*, *downsampling_ratio=None*, *padding=None*, *padding_mode='reflect'*, *transforms=None*, *return_index_spec=False*, *name=None*)

Bases: *inferno.io.core.base.SyncableDataset*

clone(*volume=None*, *transforms=None*, *name=None*)

make_sliding_windows()

pad_volume(*padding=None*)

inferno.io.volumetric.volumetric_utils module

inferno.io.volumetric.volumetric_utils.**parse_data_slice**(*data_slice*)

Parse a dataslice as a list of slice objects.

inferno.io.volumetric.volumetric_utils.**slidingwindowslices**(*shape*, *window_size*, *strides*, *ds=1*, *shuffle=True*, *rngseed=None*, *dataslice=None*, *add_overhanging=True*)

inferno.io.volumetric.volumetric_utils.**slidingwindowslices_depr**(*shape*, *nhoodsize*, *stride=1*, *ds=1*, *window=None*, *ignoreborder=True*, *shuffle=True*, *rngseed=None*, *startmins=None*, *startmaxs=None*, *dataslice=None*)

Returns a generator yielding (shuffled) sliding window slice objects. :type shape: int or list of int :param shape: Shape of the input data :type nhoodsize: int or list of int :param nhoodsize: Window size of the sliding window. :type stride: int or list of int :param stride: Stride of the sliding window. :type shuffle: bool :param shuffle: Whether to shuffle the iterator.

Module contents

Module contents

7.1.1.3 inferno.trainers package

Subpackages

inferno.trainers.callbacks package

Subpackages

inferno.trainers.callbacks.logging package

Submodules

inferno.trainers.callbacks.logging.base module

class inferno.trainers.callbacks.logging.base.**Logger**(*log_directory=None*)

Bases: *inferno.trainers.callbacks.base.Callback*

A special callback for logging.

Loggers are special because they're required to be serializable, whereas other callbacks have no such guarantees. In this regard, they jointly handled by trainers and the callback engine.

log_directory

set_log_directory(*log_directory*)

inferno.trainers.callbacks.logging.tensorboard module

class inferno.trainers.callbacks.logging.tensorboard.**TensorboardLogger**(*log_directory=None*, *log_scalars_every=None*, *log_images_every=None*, *send_image_at_batch_indice*, *send_image_at_channel_indi*, *send_volume_at_z_indices=*

Bases: *inferno.trainers.callbacks.logging.base.Logger*

Class to enable logging of training progress to Tensorboard.

Currently supports logging scalars and images.

end_of_training_iteration(**_)

end_of_validation_run(**_)

extract_images_from_batch(*batch*)

get_config()

log_histogram(*tag*, *values*, *step*, *bins=1000*)
Logs the histogram of a list/vector of values.

log_image_or_volume_batch(*tag*, *batch*, *step=None*)

log_images(*tag*, *images*, *step*)
Logs a list of images.

log_images_every

log_images_now

log_object(*tag*, *object_*, *allow_scalar_logging=True*, *allow_image_logging=True*)

log_scalar(*tag*, *value*, *step*)

tag [basestring] Name of the scalar

value step : int

training iteration

log_scalars_every

log_scalars_now

observe_state(*key*, *observe_while='training'*)

observe_states(*keys*, *observe_while='training'*)

observe_training_and_validation_state(*key*)

observe_training_and_validation_states(*keys*)

writer

Module contents

inferno.trainers.callbacks.logging.**get_logger**(*name*)

Submodules

inferno.trainers.callbacks.base module

class inferno.trainers.callbacks.base.**Callback**
Bases: object

Recommended (but not required) base class for callbacks.

bind_trainer(*trainer*)

debug_print(*message*)

get_config()

classmethod get_instances()

classmethod register_instance(*instance*)

set_config(*config_dict*)

toggle_debug()

trainer

unbind_trainer()

class inferno.trainers.callbacks.base.**CallbackEngine**
Bases: object

Gathers and manages callbacks.

Callbacks are callables which are to be called by trainers when certain events ('triggers') occur. They could be any callable object, but if endowed with a *bind_trainer* method, it's called when the callback is registered. It is recommended that callbacks (or their *__call__* methods) use the double-star syntax for keyword arguments.

BEGIN_OF_EPOCH = 'begin_of_epoch'

BEGIN_OF_FIT = 'begin_of_fit'

BEGIN_OF_SAVE = 'begin_of_save'

BEGIN_OF_TRAINING_ITERATION = 'begin_of_training_iteration'

BEGIN_OF_TRAINING_RUN = 'begin_of_training_run'

BEGIN_OF_VALIDATION_ITERATION = 'begin_of_validation_iteration'

BEGIN_OF_VALIDATION_RUN = 'begin_of_validation_run'

END_OF_EPOCH = 'end_of_epoch'

END_OF_FIT = 'end_of_fit'

END_OF_SAVE = 'end_of_save'

END_OF_TRAINING_ITERATION = 'end_of_training_iteration'

END_OF_TRAINING_RUN = 'end_of_training_run'

END_OF_VALIDATION_ITERATION = 'end_of_validation_iteration'

END_OF_VALIDATION_RUN = 'end_of_validation_run'

TRIGGERS = {'end_of_epoch', 'end_of_validation_iteration', 'begin_of_save', 'end_of_sa

bind_trainer(*trainer*)

call(*trigger*, ***kwargs*)

get_config()

rebind_trainer_to_all_callbacks()

register_callback(*callback*, *trigger='auto'*, *bind_trainer=True*)

register_new_trigger(*trigger_name*)

set_config(*config_dict*)

trainer_is_bound

unbind_trainer()

inferno.trainers.callbacks.essentials module

class inferno.trainers.callbacks.essentials.**DumpHDF5Every**(*frequency*, *to_directory*, *file-name_template='dump.{mode}.epoch{epoch_co force_dump=False*, *dump_after_every_validation_run=False*)

Bases: *inferno.trainers.callbacks.base.Callback*

Dumps intermediate training states to a HDF5 file.

add_to_dump_cache(*key*, *value*)

clear_dump_cache()

dump(*mode*)

dump_every

dump_now

dump_state(*key*, *dump_while='training'*)

dump_states(*keys*, *dump_while='training'*)

end_of_training_iteration(***_*)

end_of_validation_run(***_*)

get_file_path(*mode*)

class inferno.trainers.callbacks.essentials.**NaNDetector**

Bases: *inferno.trainers.callbacks.base.Callback*

end_of_training_iteration(***_*)

class inferno.trainers.callbacks.essentials.**ParameterEMA**(*momentum*)

Bases: *inferno.trainers.callbacks.base.Callback*

Maintain a moving average of network parameters.

apply()

end_of_training_iteration(***_*)

maintain()

class inferno.trainers.callbacks.essentials.**PersistentSave**(*template='checkpoint.pytorch.epoch{epoch_co*

Bases: *inferno.trainers.callbacks.base.Callback*

begin_of_save(***kwargs*)

end_of_save(*save_to_directory*, **_)

class inferno.trainers.callbacks.essentials.**SaveAtBestValidationScore**(*smoothness=0*, *verbose=False*)

Bases: *inferno.trainers.callbacks.base.Callback*

Triggers a save at the best EMA (exponential moving average) validation score. The basic *Trainer* has built in support for saving at the best validation score, but this callback might eventually replace that functionality.

end_of_validation_run(**_)

inferno.trainers.callbacks.scheduling module

class inferno.trainers.callbacks.scheduling.**AutoLR**(*factor*, *patience*, *required_minimum_relative_improvement=0*, *consider_improvement_with_respect_to='best'*, *cooldown_duration=None*, *monitor='auto'*, *monitor_momentum=0*, *monitor_while='auto'*, *exclude_param_groups=None*, *verbose=False*)

Bases: inferno.trainers.callbacks.scheduling._Scheduler

Callback to decay or hike the learning rate automatically when a specified monitor stops improving.

The monitor should be decreasing, i.e. lower value –> better performance.

cooldown_duration

decay()

duration_since_last_decay

duration_since_last_improvment

end_of_training_iteration(**_)

end_of_validation_run(**_)

in_cooldown

static is_significantly_less_than(*x*, *y*, *min_relative_delta*)

maintain_monitor_moving_average()

monitor_value_has_significantly_improved

out_of_patience

patience

class inferno.trainers.callbacks.scheduling.**AutoLRDecay**(*factor*, *patience*, *required_minimum_relative_improvement=0*, *consider_improvement_with_respect_to='best'*, *cooldown_duration=None*, *monitor='auto'*, *monitor_momentum=0*, *monitor_while='auto'*, *exclude_param_groups=None*, *verbose=False*)

Bases: *inferno.trainers.callbacks.scheduling.AutoLR*

Callback to decay the learning rate automatically when a specified monitor stops improving.

The monitor should be decreasing, i.e. lower value –> better performance.

class inferno.trainers.callbacks.scheduling.**DecaySpec**(*duration*, *factor*)

Bases: object

A class to specify when to decay (or hike) LR and by what factor.

classmethod build_from(*args*)

match(*iteration_count=None*, *epoch_count=None*, *when_equal_return=True*)

new()

class inferno.trainers.callbacks.scheduling.**ManualLR**(*decay_specs*, *exclude_param_groups=None*)

Bases: *inferno.trainers.callbacks.base.Callback*

decay(*factor*)

end_of_training_iteration(**_)

match()

Module contents

Submodules

inferno.trainers.basic module

class inferno.trainers.basic.**Trainer**(*model=None*)

Bases: object

A basic trainer.

Given a torch model, this class encapsulates the training and validation loops, checkpoint creation, logging, CPU <-> GPU transfers and managing data-loaders.

In addition, this class interacts with the callback engine (found at *inferno.trainers.callbacks.base.CallbackEngine*), which manages callbacks at certain preset events.

Notes

Logging is implemented as a special callback, in the sense that it's jointly managed by the this class and the callback engine. This is primarily because general callbacks are not intended to be serializable, but not being able to serialize the logger is a nuisance.

DYNAMIC_STATES = {'learning_rate': 'current_learning_rate'}

INF_STRINGS = {'inf', 'infinity', 'infty'}

apply_model(**inputs*)

apply_model_and_loss(*inputs*, *target*, *backward=True*)

bind_loader(*name*, *loader*, *num_inputs=None*, *num_targets=1*)
Bind a data loader to the trainer.

Parameters

- **name** (*{'train', 'validate', 'test'}*) – Name of the loader, i.e. what it should be used for.
- **loader** (*torch.utils.data.DataLoader*) – DataLoader object.
- **num_inputs** (*int*) – Number of input tensors from the *loader*.
- **num_targets** (*int*) – Number of target tensors from the *loader*.

Returns self

Return type *Trainer*

Raises

- KeyError – if name is invalid.
- TypeError – if loader is not a DataLoader instance.

bind_model(*model*)
Binds a model to the trainer. Equivalent to setting model.

Parameters **model** (*torch.nn.Module*) – Model to bind.

Returns self.

Return type *Trainer*

classmethod build(*model=None*, ***trainer_config*)
Factory function to build the trainer.

build_criterion(*method*, ***kwargs*)
Builds the loss criterion for training.

Parameters

- **method** (*str or callable or torch.nn.Module*) – Name of the criterion when str, criterion class when callable, or a torch.nn.Module instance. If a name is provided, this method looks for the criterion in *torch.nn*.
- **kwargs** (*dict*) – Keyword arguments to the criterion class' constructor if applicable.

Returns self.

Return type *Trainer*

Raises

- AssertionError – if criterion is not found.
- NotImplementedError – if method is neither a str nor a callable.

build_logger(*logger=None*, *log_directory=None*, ***kwargs*)
Build the logger.

Parameters

- **logger** (*inferno.trainers.callbacks.logging.base.Logger or str or type*) – Must either be a Logger object or the name of a logger or the class of a logger.
- **log_directory** (*str*) – Path to the directory where the log files are to be stored.
- **kwargs** (*dict*) – Keyword arguments to the logger class.

Returns self

Return type *Trainer*

build_metric(*method*, ***kwargs*)
Builds the metric for evaluation.

Parameters

- **method** (*callable or str*) – Name of the metric when string, metric class or a callable object when callable. If a name is provided, this method looks for the metric in *inferno.extensions.metrics*.
- **kwargs** (*dict*) – Keyword arguments to the metric class' constructor, if applicable.

Returns self.

Return type *Trainer*

Raises AssertionError: if the metric is not found.

build_optimizer(*method*, *param_groups=None*, ***kwargs*)
Builds the optimizer for training.

Parameters

- **method** (*str or callable or torch.optim.Optimizer*) – Name of the optimizer when str, handle to the optimizer class when callable, or a torch.optim.Optimizer instance. If a name is provided, this method looks for the optimizer in *torch.optim* module first and in inferno.extensions.optimizers second.
- **param_groups** (*list of dict*) – Specifies the parameter group. Defaults to model.parameters() if None.
- **kwargs** (*dict*) – Keyword arguments to the optimizer.

Returns self.

Return type *Trainer*

Raises

- `AssertionError` – if optimizer is not found
- `NotImplementedError` – if method is not str or callable.

callbacks
Gets the callback engine.

cast(*objects*)

console
Get the current console.

cpu()
Train on the CPU.

Returns self

Return type *Trainer*

criterion
Gets the loss criterion.

criterion_is_defined

cuda(*devices=None*, *base_device=None*)
Train on the GPU.

Parameters

- **devices** (*list*) – Specify the ordinals of the devices to use for dataparallel training.
- **base_device** (*{'cpu', 'cuda'}*) – When using data-parallel training, specify where the result tensors are collected. If 'cuda', the results are collected in *devices[0]*.

Returns self

Return type *Trainer*

current_learning_rate

dtype

epoch_count

eval_mode()
Set model, criterion and metric to eval mode

evaluate_metric_every(*frequency*)
Set frequency of metric evaluation __during **training**__ (and not during validation).

Parameters **frequency** (*inferno.utils.train_utils.Frequency or str or tuple or list or int*) – Metric evaluation frequency. If str, it could be (say) '10 iterations' or '1 epoch'. If tuple (or list), it could be (10, 'iterations') or (1, 'epoch'). If int (say 10), it's interpreted as (10, 'iterations').

Returns self

Return type *Trainer*

evaluate_metric_now

evaluating_metric_every

fetch_next_batch(*from_loader='train'*, *restart_exhausted_generators=True*, *update_batch_count=True*, *update_epoch_count_if_generator_exhausted=True*)

fit(*max_num_iterations=None*, *max_num_epochs=None*)
Fit model.

Parameters

- **max_num_iterations** (*int or float or str*) – (Optional) Maximum number of training iterations. Overrides the value set by *Trainer.set_max_num_iterations*. If float, it should equal numpy.inf. If str, it should be one of {'inf', 'infinity', 'infty'}.
- **max_num_epochs** (*int or float or str*) – (Optional) Maximum number of training epochs. Overrides the value set by *Trainer.set_max_num_epochs*. If float, it should equal numpy.inf. If str, it should be one of {'inf', 'infinity', 'infty'}.

Returns self

Return type *Trainer*

get_config(*exclude_loader=True*)

get_current_learning_rate()
Gets the current learning rate. :returns: List of learning rates if there are multiple parameter groups, or a float

if there's just one.

Return type list or float

get_loader_specs(*name*)

get_state(*key*, *default=None*)

is_cuda()
Returns whether using GPU for training.

iteration_count

load(*from_directory=None*, *best=False*, *filename=None*)
Load the trainer from checkpoint.

Parameters

- **from_directory** (*str*) – Path to the directory where the checkpoint is located. The filename should be 'checkpoint.pytorch' if best=False, or 'best_checkpoint.pytorch' if best=True.
- **best** (*bool*) – Whether to load the best checkpoint. The filename in *from_directory* should be 'best_checkpoint.pytorch'.
- **filename** (*str*) – Overrides the default filename.

Returns self

Return type *Trainer*

load_(**args*, ***kwargs*)

load_model(*from_directory=None*, *filename=None*)

log_directory
Gets the log directory.

logger
Gets the logger.

metric
Gets the evaluation metric.

metric_is_defined
Checks if the metric is defined.

model
Gets the model.

model_is_defined

next_epoch()

next_iteration()

optimizer
Gets the optimizer.

optimizer_is_defined

print(*message*)

record_validation_results(*validation_loss*, *validation_error*)

register_callback(*callback*, *trigger='auto'*, ***callback_kwargs*)
Registers a callback with the internal callback engine.

Parameters

- **callback** (*type or callable*) – Callback to register.
- **trigger** (*str*) – Specify the event that triggers the callback. Leave at 'auto' to have the callback-engine figure out the triggers. See *inferno.training.callbacks.base.CallbackEngine* documentation for more on this.
- **callback_kwargs** (*dict*) – If *callback* is a type, initialize an instance with these keywords to the __init__ method.

Returns self.

Return type *Trainer*

restart_generators(*of_loader=None*)

save(*exclude_loader=True*, *stash_best_checkpoint=True*)

save_at_best_validation_score(*yes=True*)
Sets whether to save when the validation score is the best seen.

save_directory

save_every(*frequency*, *to_directory=None*, *checkpoint_filename=None*, *best_checkpoint_filename=None*)
Set checkpoint creation frequency.

Parameters

- **frequency** (*inferno.utils.train_utils.Frequency or tuple or str*) – Checkpoint creation frequency. Examples: '100 iterations' or '1 epochs'.
- **to_directory** (*str*) – Directory where the checkpoints are to be created.
- **checkpoint_filename** (*str*) – Name of the checkpoint file.
- **best_checkpoint_filename** (*str*) – Name of the best checkpoint file.

Returns self.

Return type *Trainer*

save_model(*to_directory=None*)

save_now

save_to_directory(*to_directory=None*, *checkpoint_filename=None*, *best_checkpoint_filename=None*)

saving_every
Gets the frequency at which checkpoints are made.

set_config(*config_dict*)

set_log_directory(*log_directory*)
Set the directory where the log files are to be stored.

Parameters **log_directory** (*str*) – Directory where the log files are to be stored.

Returns self

Return type *Trainer*

set_max_num_epochs(*max_num_epochs*)
Set the maximum number of training epochs.

Parameters **max_num_epochs** (*int or float or str*) – Maximum number of training epochs. If float, it should equal numpy.inf. If str, it should be one of {'inf', 'infinity', 'infty'}.

Returns self

Return type *Trainer*

set_max_num_iterations(*max_num_iterations*)
Set the maximum number of training iterations.

Parameters **max_num_iterations** (*int or float or str*) – Maximum number of training iterations. If float, it should equal numpy.inf. If str, it should be one of {'inf', 'infinity', 'infty'}.

Returns self

Return type *Trainer*

set_precision(*dtype*)
Set training precision.

Parameters **dtype** (*{'double', 'float', 'half'}*) – Training precision.

Returns self

Return type *Trainer*

set_target_batch_dim(*value*)

split_batch(*batch*, *from_loader*)

stop_fitting(*max_num_iterations=None*, *max_num_epochs=None*)

target_batch_dim

to_device(*objects*)

train_for(*num_iterations=None*, *break_callback=None*)

train_loader

train_mode()
Set model, criterion and metric to train mode

update_state(*key*, *value*)

update_state_from_dictionary(*dictionary*)

update_state_from_model_state_hooks()

validate_every(*frequency*, *for_num_iterations=None*)
Set validation frequency.

Parameters

- **frequency** (*inferno.utils.train_utils.Frequency or str or tuple or list or int*) – Validation frequency. If str, it could be (say) '10 iterations' or '1 epoch'. If tuple (or list), it could be (10, 'iterations') or (1, 'epoch'). If int (say 10), it's interpreted as (10, 'iterations').
- **for_num_iterations** (*int*) – Number of iterations to validate for. If not set, the model is validated on the entire dataset (i.e. till the data loader is exhausted).

Returns self

Return type *Trainer*

validate_for(*num_iterations=None*, *loader_name='validate'*)
Validate for a given number of validation (if *num_iterations is not None*) or over the entire (validation) data set.

Parameters

- **num_iterations** (*int*) – Number of iterations to validate for. To validate on the entire dataset, leave this as *None*.
- **loader_name** (*str*) – Name of the data loader to use for validation. 'validate' is the obvious default.

Returns self.

Return type *Trainer*

validate_loader

validate_now

validating_every

verify_batch(*batch*, *from_loader*)

wrap_batch(*batch*, *from_loader=None*, *requires_grad=False*, *volatile=False*)

Module contents

7.1.1.4 inferno.utils package

Submodules

inferno.utils.exceptions module

Exceptions and Error Handling

exception inferno.utils.exceptions.**ClassNotFoundError**
Bases: LookupError

exception inferno.utils.exceptions.**DTypeError**
Bases: TypeError

exception inferno.utils.exceptions.**DeviceError**
Bases: ValueError

exception inferno.utils.exceptions.**FrequencyTypeError**
Bases: TypeError

exception inferno.utils.exceptions.**FrequencyValueError**
Bases: ValueError

exception inferno.utils.exceptions.**NotSetError**
Bases: ValueError

exception inferno.utils.exceptions.**NotTorchModuleError**
Bases: TypeError

exception inferno.utils.exceptions.**NotUnwrappableError**
Bases: NotImplementedError

exception inferno.utils.exceptions.**ShapeError**
Bases: ValueError

inferno.utils.exceptions.**assert_**(*condition*, *message=''*, *exception_type=<class 'AssertionError'>*)
Like assert, but with arbitrary exception types.

inferno.utils.io_utils module

inferno.utils.io_utils.**fromh5**(*path*, *datapath=None*, *dataslice=None*, *asnumpy=True*, *pretrain=None*)
Opens a hdf5 file at path, loads in the dataset at datapath, and returns dataset as a numpy array.

inferno.utils.io_utils.**print_tensor**(*tensor*, *prefix*, *directory*)
Prints a image or volume tensor to file as images.

inferno.utils.io_utils.**toh5**(*data*, *path*, *datapath='data'*, *compression=None*, *chunks=None*)
Write *data* to a HDF5 volume.

inferno.utils.io_utils.**yaml2dict**(*path*)

inferno.utils.model_utils module

class inferno.utils.model_utils.**ModelTester**(*input_shape*, *expected_output_shape*)
Bases: object

cuda()

get_input()

inferno.utils.model_utils.**is_model_cuda**(*model*)

inferno.utils.python_utils module

Utility functions with no external dependencies.

inferno.utils.python_utils.**as_tuple_of_len**(*x*, *len_*)

class inferno.utils.python_utils.**delayed_keyboard_interrupt**
Bases: object

Delays SIGINT over critical code. Borrowed from: https://stackoverflow.com/questions/842557/ how-to-prevent-a-block-of-code-from-being-interrupted-by-keyboardinterrupt-in-py

handler(*sig*, *frame*)

inferno.utils.python_utils.**deprecated**(*reason*)
This is a decorator which can be used to mark functions as deprecated. It will result in a warning being emitted when the function is used.

Borrowed from https://stackoverflow.com/questions/2536307/ decorators-in-the-python-standard-lib-deprecated-specifically by Laurent LAPORTE https://stackoverflow.com/users/1513933/laurent-laporte

inferno.utils.python_utils.**from_iterable**(*x*)

inferno.utils.python_utils.**get_config_for_name**(*config*, *name*)

inferno.utils.python_utils.**has_callable_attr**(*object_*, *name*)

inferno.utils.python_utils.**is_listlike**(*x*)

inferno.utils.python_utils.**is_maybe_list_of**(*check_function*)

inferno.utils.python_utils.**robust_len**(*x*)

inferno.utils.python_utils.**to_iterable**(*x*)

inferno.utils.test_utils module

inferno.utils.test_utils.**generate_random_data**(*num_samples*, *shape*, *num_classes*, *hardness=0.3*, *dtype=None*)
Generate a random dataset with a given hardness and number of classes.

inferno.utils.test_utils.**generate_random_dataloader**(*num_samples*, *shape*, *num_classes*, *hardness=0.3*, *dtype=None*, *batch_size=1*, *shuffle=False*, *num_workers=0*, *pin_memory=False*)
Generate a loader with a random dataset of given hardness and number of classes.

inferno.utils.test_utils.**generate_random_dataset**(*num_samples*, *shape*, *num_classes*, *hardness=0.3*, *dtype=None*)
Generate a random dataset with a given hardness and number of classes.

inferno.utils.torch_utils module

inferno.utils.torch_utils.**assert_same_size**(*tensor_1*, *tensor_2*)

inferno.utils.torch_utils.**flatten_samples**(*tensor_or_variable*)
Flattens a tensor or a variable such that the channel axis is first and the sample axis is second. The shapes are transformed as follows:

(N, C, H, W) –> (C, N * H * W) (N, C, D, H, W) –> (C, N * D * H * W) (N, C) –> (C, N)

The input must be atleast 2d.

inferno.utils.torch_utils.**is_image_or_volume_tensor**(*object_*)

inferno.utils.torch_utils.**is_image_tensor**(*object_*)

inferno.utils.torch_utils.**is_label_image_or_volume_tensor**(*object_*)

inferno.utils.torch_utils.**is_label_image_tensor**(*object_*)

inferno.utils.torch_utils.**is_label_tensor**(*object_*)

inferno.utils.torch_utils.**is_label_volume_tensor**(*object_*)

inferno.utils.torch_utils.**is_matrix_tensor**(*object_*)

inferno.utils.torch_utils.**is_scalar_tensor**(*object_*)

inferno.utils.torch_utils.**is_tensor**(*object_*)

inferno.utils.torch_utils.**is_volume_tensor**(*object_*)

inferno.utils.torch_utils.**unwrap**(*tensor_or_variable*, *to_cpu=True*, *as_numpy=False*)

inferno.utils.torch_utils.**where**(*condition*, *if_true*, *if_false*)
Torch equivalent of numpy.where.

Parameters

- **condition** (*torch.ByteTensor or torch.cuda.ByteTensor or torch.autograd.Variable*) – Condition to check.
- **if_true** (*torch.Tensor or torch.cuda.Tensor or torch.autograd.Variable*) – Output value if condition is true.
- **if_false** (*torch.Tensor or torch.cuda.Tensor or torch.autograd.Variable*) – Output value if condition is false

Returns

Return type torch.Tensor

Raises

- `AssertionError` – if if_true and if_false are not both variables or both tensors.
- `AssertionError` – if if_true and if_false don't have the same datatype.

inferno.utils.train_utils module

Utilities for training.

class inferno.utils.train_utils.**AverageMeter**
Bases: `object`

Computes and stores the average and current value. Taken from https://github.com/pytorch/examples/blob/master/imagenet/main.py

reset()

update(*val*, *n=1*)

class inferno.utils.train_utils.**CLUI**
Bases: `object`

Command Line User Interface

class inferno.utils.train_utils.**Duration**(*value=None*, *units=None*)
Bases: *inferno.utils.train_utils.Frequency*

Like frequency, but measures a duration.

compare(*iteration_count=None*, *epoch_count=None*)

match(*iteration_count=None*, *epoch_count=None*, *when_equal_return=False*, **_)

class inferno.utils.train_utils.**Frequency**(*value=None*, *units=None*)
Bases: `object`

UNIT_PRIORITY = 'iterations'

VALID_UNIT_NAME_MAPPING = {'epoch': 'epochs', 'epochs': 'epochs', 'iteration': 'ite

assert_units_consistent(*units=None*)

assert_value_consistent(*value=None*)

classmethod build_from(*args*, *priority='iterations'*)

by_epoch

by_iteration

epoch()

every(*value*)

classmethod from_string(*string*)

is_consistent

iteration()

match(*iteration_count=None*, *epoch_count=None*, *persistent=False*, *match_zero=True*)

units

value

class inferno.utils.train_utils.**MovingAverage**(*momentum=0*)
Bases: object

Computes the moving average of a given float.

relative_change

reset()

update(*val*)

class inferno.utils.train_utils.**NoLogger**(*logdir=None*)
Bases: object

log_value(**kwargs*)

inferno.utils.train_utils.**get_state**(*module*, *key*, *default=None*)
Gets key from *module*'s state hooks.

inferno.utils.train_utils.**set_state**(*module*, *key*, *value*)
Writes *key-value* pair to *module*'s state hook.

Module contents

7.1.2 Submodules

7.1.3 inferno.inferno module

Main module.

7.1.4 Module contents

Top-level package for inferno.

CHAPTER 8

Credits

8.1 Development Lead

- Nasim Rahaman @ Image Analysis and Learning Lab , Heidelberg Collaboratory for Image Processing ,

8.2 Contributors

In no particular order,

- Steffen Wolf @ Image Analysis and Learning Lab , Heidelberg Collaboratory for Image Processing ,
- Maurice Weiler @ Amsterdam Machine Learning Lab , University of Amsterdam ,
- Constantin Pape @ Image Analysis and Learning Lab , Heidelberg Collaboratory for Image Processing ,
- Sven Peter @ Image Analysis and Learning Lab , Heidelberg Collaboratory for Image Processing ,
- Manuel Haussmann @ Image Analysis and Learning Lab , Heidelberg Collaboratory for Image Processing ,
- Thorsten Beier @ Image Analysis and Learning Lab , Heidelberg Collaboratory for Image Processing ,
- Benjamin Striner @ Machine Learning Department , Carnegie Mellon University ,

CHAPTER 9

History

9.1 0.1.0 (2017-08-24)

- First early release on PyPI

9.2 0.1.1 (2017-08-24)

- Version Increment

9.3 0.1.2 (2017-08-24)

- Version Increment

9.4 0.1.3 (2017-08-24)

- Updated Documentation

9.5 0.1.4 (2017-08-24)

- travis auto-deployment on pypi

9.6 0.1.5 (2017-08-24)

- travis changes to run unittest

9.7 0.1.6 (2017-08-24)

- travis missing packages for unittesting
- fixed inconsistent version numbers

9.8 0.1.7 (2017-08-25)

- setup.py critical bugix in install procedure

CHAPTER 10

Bibliography

The bibliography:

Top-level package for inferno.

CHAPTER 11

Indices and tables

- genindex
- modindex
- search

Python Module Index

i

Index

C

D

E

F

G

K

L

M

N

O

P

R

S

T

U

V

W

Y

Z