easycore

Mar 24, 2020

Contents

CHAPTER 1

Tutorials

1.1 Installation

1.1.1 Requirements

- Python 3.6
- PyTorch (optional, for `easycore.torch` package)

1.1.2 Install

Install from PyPI

```
pip install easycore
```

Install from GitHub

```
pip install 'git+https://github.com/YuxinZhaozyx/easycore.git'
```

Install from a local clone

```
git clone https://github.com/YuxinZhaozyx/easycore.git
cd easycore
pip install -e .
```

1.2 Light weight config tools

easycore make it easy to load config from local yaml file, save config and control the config in runtime.

1.2.1 Load config from local yaml file

An example of yaml file is shown bellow:

```yaml
MODEL:
    IN_FEAUTRES: ["res3", "res4", "res5"]
    INPUT_SIZE: (224, 224)
    NUM_CLASSES: 100
NAME: YuxinZhaozyx
```

You can load the yaml file in the follow way:

```python
from easycore.common.config import CfgNode as CN

cfg = CN.open('example.yaml')

# or
with open('example.yaml', 'r', encoding='utf-8') as f:
    cfg = CN.open(f)
```

1.2.2 Get an empty config

```python
cfg = CN()
```

1.2.3 Get a config from from python `dict`

```python
init_dict = {
    "MODEL": {
        "IN_FEATURES": ["res3", "res4", "res5"],
        "INPUT_SIZE": (224, 224),
        "NUM_CLASSES": 100,
    },
    "NAME": "YuxinZhaozyx",
}
cfg = CN(init_dict)
```

1.2.4 Use config

```python
# get value from config
# the config has been automatically transform into python data type.
in_features = cfg.MODEL.IN_FEATURES  # list
input_size = cfg.MODEL.INPUT_SIZE    # tuple
num_classes = cfg.MODEL.NUM_CLASSES  # int
name = cfg.NAME                      # str
```

(continues on next page)

(continued from previous page)

```
# add new value to config
cfg.LICENSE = 'MIT'

# add a new CfgNode to config
cfg.SOLVER = CN()
cfg.SOLVER.LEARNING_RATE = 0.001
cfg.SOLVER.BATCH_SIZE = 128
```

1.2.5 Merge two config

```
cfg_a = CN()
cfg_a.key1 = 1
cfg_a.key2 = 2

cfg_b = CN()
cfg_b.key2 = 3
cfg_c.key3 = 4

# merge two config
cfg_a.merge(cfg_b)  # now cfg_a.key2 is 3
```

1.2.6 Copy a config

```
cfg_copy = cfg.copy()  # get a deepcopy of cfg
```

1.2.7 Save config to yaml file

```
cfg.save("example-save.yaml")

# or
with open("example-save.yaml", 'w', encoding='utf-8') as f:
    cfg.save(f)
```

1.2.8 API Documentation

- easycore.common.config

1.3 Multiprocessing parallel acceleration tools

easycore make it easy to parallel your tasks in cpus and gpus.

1.3.1 API

You can write a parallel runner by inheriting class `UnorderedRunner` or `OrderedRunner` and overriding following 6 static methods.

```
@staticmethod
def producer_init(device, cfg):
    """
    function for producer initialization.

    Args:
        device (str): device for the this process.
        cfg (easycore.common.config.CfgNode): config of this process, you can use it␣
↪to transfer data
            to `producer_work` and `producer_end` function.
    """
    pass

@staticmethod
def producer_work(device, cfg, data):
    """
    function specify how the producer processes the data.

    Args:
        device (str): device for this process.
        cfg (easycore.common.config.CfgNode): config of this process, you can use it␣
↪to get data from
            `producer_init` function and transfer data to the next `producer_work`␣
↪and `producer_end`
            function.
        data (Any): data get from input of `__call__` method.

    Returns:
        Any: processed data
    """
    return data

@staticmethod
def producer_end(device, cfg):
    """
    function after finishing all of its task and before close the process.

    Args:
        device (str): device for this process.
        cfg (easycore.common.config.CfgNode): config of this process, you can use it␣
↪to get data
            from `producer_init` and `producer_work` function.
    """
    pass

@staticmethod
def consumer_init(cfg):
    """
    function for consumer initialization.

    Args:
        cfg (easycore.common.config.CfgNode): config of this process, you can use it␣
↪to transfer data
            to `consumer_work` and `consumer_end` function.
    """
    pass
```

(continues on next page)

(continued from previous page)

```
@staticmethod
def consumer_work(cfg, data):
    """
    function specify how the consumer processses the data from producers.

    Args:
        cfg (easycore.common.config.CfgNode): config of this process, you can use it␣
↪to get data from
            `consumer_init` function and transfer data to the next `consumer_work`␣
↪and `consumer_end`
            function.
    """
    pass

@staticmethod
def consumer_end(cfg):
    """
    function after receiving all data from producers.

    Args:
        cfg (easycore.common.config.CfgNode): config of this process, you can use it␣
↪get data from
            `consumer_work` function.

    Returns:
        Any: processed data
    """
    return None
```

1.3.2 Example 1: Sum of squares

It can be implemented with a simple way:

```
data_list = list(range(100))
result = sum([data * data for data in data_list])

# or more simple
result = 0
for data in data_list:
    square = data * data
    result += square
```

We calculate square of each element of the list, and then sum they together. In this case, it can be divided into two tasks. We assign this two tasks to producer and consumer respectively.

```
from easycore.common.config import CfgNode
from easycore.common.parallel import UnorderedRunner

class Runner(UnorderedRunner):
    @staticmethod
    def producer_work(device, cfg, data):
        return data * data  # calculate square of data

    @staticmethod
    def consumer_init(cfg):
```

(continues on next page)

(continued from previous page)

```
        cfg.sum = 0  # init a sum variable with 0, you can use cfg to transfer data

    @staticmethod
    def consumer_work(cfg, data):
        cfg.sum += data  # add the square to the sum variable

    @staticmethod
    def consumer_end(cfg):
        return cfg.sum  # return the result you need

if __name__ == '__main__':
    runner = Runner(devices=3)  # if you specify `device with a integer`, it will use␣
↪cpus.
    # You can specify a list of str instead, such as:
    # runner = Runner(devices=["cpu", "cpu", "cpu"])

    data_list = list(range(100))  # prepare data, it must be iterable
    result = runner(data_list)  # call the runner
    print(result)

    runner.close()  # close the runner and shutdown all processes it opens.
```

1.3.3 Example 2: An neural network predictor

First we define an neural network in `network.py`:

```
import torch
import torch.nn as nn
import torch.nn.functional as F

class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.fc = nn.Linear(1, 3)

    def forward(self, x):
        x = self.fc(x)
        x = F.relu(x)
        return x
```

The network can be paralleled to 4 gpus in the following way:

```
from easycore.common.config import CfgNode
from easycore.common.parallel import OrderedRunner
from network import Net
import torch

class Predictor(OrderedRunner):
    @staticmethod
    def producer_init(device, cfg):
        cfg.model = Net()      # init the producer with a model
        cfg.model.to(device)   # transfer the model to certain device

    @staticmethod
    def producer_work(device, cfg, data):
```

(continues on next page)

(continued from previous page)

```
        with torch.no_grad():
            data = torch.Tensor([[data]])  # preprocess data
            data = data.to(device)  # transfer data to certain device
            output = cfg.model(data)  # predict
            output = output.cpu()  # transfer result to cpu
        return output

    @staticmethod
    def producer_end(device, cfg):
        del cfg.model  # delete the model when all data has been predicted.

    @staticmethod
    def consumer_init(cfg):
        cfg.data_list = []  # prepare a list to store all data from producers.

    @staticmethod
    def consumer_work(cfg, data):
        cfg.data_list.append(data)  # store data from producers.

    @staticmethod
    def consumer_end(cfg):
        data = torch.cat(cfg.data_list, dim=0)  # postprocess data.
        return data

if __name__ == '__main__':
    predictor = Predictor(devices=["cuda:0", "cuda:1", "cuda:2", "cuda:3"])  # init a␣
↪parallel predictor

    data_list = list(range(100))  # prepare data
    result = predictor(data_list)  # predict
    print(result.shape)

    predictor.close()  # close the predictor when you no longer need it.
```

1.3.4 Example 3: Process data with batch

You can use a simple generator or pytorch dataloader to generate batch data.

```
from easycore.common.config import CfgNode
from easycore.torch.parallel import OrderedRunner
from network import Net
import torch

def batch_generator(data_list, batch_size):
    for i in range(0, len(data_list), batch_size):
        data_batch = data_list[i : i+batch_size]
        yield data_batch

class Predictor(OrderedRunner):

    @staticmethod
    def producer_init(device, cfg):
        cfg.model = Net()
        cfg.model.to(device)
```

(continues on next page)

(continued from previous page)

```
    @staticmethod
    def producer_work(device, cfg, data):
        with torch.no_grad():
            data = torch.Tensor(data).view(-1,1)
            data = data.to(device)
            output = cfg.model(data)
            output = output.cpu()
        return output

    @staticmethod
    def producer_end(device, cfg):
        del cfg.model

    @staticmethod
    def consumer_init(cfg):
        cfg.data_list = []

    @staticmethod
    def consumer_work(cfg, data):
        cfg.data_list.append(data)

    @staticmethod
    def consumer_end(cfg):
        data = torch.cat(cfg.data_list, dim=0)
        return data

if __name__ == '__main__':
    predictor = Rredictor(devices=["cuda:0", "cuda:1"])

    data_list = list(range(100))
    result = predictor(batch_generator(data_list, batch_size=10))
    print(result.shape)

    predictor.close()
```

Here, we replace `easycore.common.parallel` with `easycore.torch.parallel`. `easycore.torch.parallel` has the same API with `easycore.common.parallel` but use `torch.multiprocessing` library instead of `multiprocessing` library.

1.3.5 Example 4: Transfer outside parameters into Runner

You can transfer parameters into runner through `cfg` parameter. `cfg` is a `easycore.common.config.CfgNode`. See tutorial "Light weight config tools" for how to use it.

We use "sum of power" as an example:

```
from easycore.common.config import CfgNode as CN
from easycore.common.parallel import UnorderedRunner

class Runner(UnorderedRunner):
    @staticmethod
    def producer_work(device, cfg, data):
        return data ** cfg.exponent  # calculate power of data with outside parameter
↪"exponent".
```

(continues on next page)

(continued from previous page)

```
    @staticmethod
    def consumer_init(cfg):
        cfg.sum = 0  # init a sum variable with 0, you can use cfg to transfer data

    @staticmethod
    def consumer_work(cfg, data):
        cfg.sum += data  # add the square to the sum variable

    @staticmethod
    def consumer_end(cfg):
        return cfg.sum  # return the result you need

if __name__ == '__main__':
    # set parameters outside.
    cfg = CN()
    cfg.exponent = 3

    runner = Runner(devices=3, cfg=cfg)  # transfer `cfg` into the runner

    data_list = list(range(100))
    result = runner(data_list)
    print(result)

    runner.close()
```

1.3.6 API Documentation

- easycore.common.parallel
- easycore.torch.parallel

1.4 Register Mechanism

easycore make it easy to register an object with name, and get it later.

1.4.1 Create a registry

```
MODEL_REGISTRY = Registry("MODEL")
```

1.4.2 Register an object with its `__name__`

```
@MODEL_REGISTRY.register()
class ResNet50:
    pass

# or

MODEL_REGISTRY.register(obj=ResNet50)
```

1.4.3 Register an object with a given name

```
@MODEL_REGISTRY.register("resnet")
class RestNet50:
    pass

# or

MODEL_REGISTRY.register("resnet", ResNet50)
```

1.4.4 Get a registered object from registry

```
model_class = MODEL_REGISTRY.get("ResNet50")

# or

model_class = MODEL_REGISTRY.get("resnet")
```

1.4.5 API Documentation

- easycore.common.registry

CHAPTER 2

API Documentation

2.1 easycore.common

2.1.1 easycore.common.config

class easycore.common.config.**CfgNode**(*init_dict: dict = None, copy=True*)

Bases: *dict*

Config Node

__init__(*init_dict: dict = None, copy=True*)

Parameters

- **init_dict** (*dict*) – a possibly-nested dictionary to initialize the CfgNode.
- **copy** (*bool*) – if this option is set to False, the CfgNode instance will share the value with the *init_dict*, otherwise the contents of *init_dict* will be deepcopied.

freeze(*frozen: bool = True*)

freeze or unfreeze the CfgNode and all of its children

Parameters **frozen** (*bool*) – freeze or unfreeze the config

is_frozen()

get the state of the config.

Returns *bool* – whether the config tree is frozen.

copy()

deepcopy this CfgNode

Returns *CfgNode*

merge(*cfg*)

merge another CfgNode into this CfgNode, the another CfgNode will override this CfgNode.

Parameters **cfg** (*CfgNode*) –

save(*save_path*, *encoding='utf-8'*)
save the CfgNode into a yaml file

Parameters **save_path** –

classmethod open(*file*, *encoding='utf-8'*)
load a CfgNode from file.

Parameters

- **file** (*io.IOBase or str*) – file object or path to the yaml file.
- **encoding** (*str*) –

Returns *CfgNode*

classmethod load(*yaml_str: str*)
load a CfgNode from a string of yaml format

Parameters **yaml_str** (*str*) –

Returns *CfgNode*

classmethod dump(*cfg*, *stream=None*, *encoding=None*, ***kwargs*)
dump CfgNode into yaml str or yaml file

Note: if *stream* option is set to non-None object, the CfgNode will be dumpped into stream and return None, if *stream* option is not given or set to None, return a string instead.

Parameters

- **cfg** (CfgNode) –
- **stream** (*io.IOBase or None*) – if set to a file object, the CfgNode will be dumpped into stream and return None, if set to None, return a string instead.
- **encoding** (*str or None*) –
- ****kwargs** – options of the yaml dumper.

  Some useful options: ["allow_unicode", "line_break", "explicit_start", "explicit_end", "version", "tags"].

  See more details at https://github.com/yaml/pyyaml/blob/2f463cf5b0e98a52bc20e348d1e69761bf263b86/lib3/yaml/__init__.py#L252

Returns *None or str*

dict()
convert to a dict

Returns *dict*

__str__()

Returns *str* – a str of dict format

class easycore.common.config.**HierarchicalCfgNode**
Bases: object

Config Node help class for open yaml file that depends on another yaml file.

You can specify the dependency between yaml files with __BASE__ tag.

Example

We can load yaml file `example-A.yaml` which depends on `example-B.yaml` in the following way.

`example-A.yaml`:

```
__BASE__: ./example-B.yaml
A: in example-A.yaml
C: in example-A.yaml
```

`example-B.yaml`:

```
A: in example-B.yaml
B: in example-B.yaml
```

Now, you can open example-A.yaml:

```
>>> import easycore.common.config import HierarchicalCfgNode
>>> cfg = HierarchicalCfgNode.open("./example-A.yaml")
>>> print(cfg)
{"A" : "in example-A.yaml", "B" : "in example-B.yaml", "C" : "in example-A.yaml"}
```

Attributes in `example-A.yaml` will cover attributes in `example-B.yaml`.

Note: `__BASE__` can be an absolute path or a path relative to the yaml file. And it will be first considered as a path relative to the yaml file then an absolute path.

classmethod open(*file*, *encoding='utf-8'*)

load a CfgNode from file.

Parameters

- **file** (*str*) – path to the yaml file.
- **encoding** (*str*) –

Returns *CfgNode*

classmethod save(*cfg*, *save_path*, *base_cfg_path=None*, *base_path_relative=True*, *encoding='utf-8'*)

save the CfgNode into a yaml file

Parameters

- **cfg** (CfgNode) –
- **save_path** (*str*) –
- **base_cfg_path** (*str*) – if not specified, it behavior like `cfg.save(save_path, encoding)`.
- **base_path_relative** (*bool*) – whether to set base cfg path to a path relative to the save_path.
- **encoding** (*str*) –

2.1.2 easycore.common.parallel

class easycore.common.parallel.**BaseRunner**(*devices*, *cfg={}*, *queue_scale=3.0*)

Bases: object

A Multi-process runner whose consumer receive data in unorder. The runner will start multi-processes for producers and 1 thread for consumer.

__init__(*devices*, *cfg={}*, *queue_scale=3.0*)

Parameters

- **devices** (*int or Iterable*) – If the *devices* is *int*, it will use devices cpu to do the work. If the *devices* is an iterable object, such as list, it will use the devices specified by the iterable object, such as ["cpu", "cuda:0", "cuda:1"].
- **cfg** (easycore.common.config.CfgNode) – user custom data.
- **queue_scale** (*float*) – scale the queues for communication between processes.

is_activate

whether the runner is alive.

static producer_init(*device*, *cfg*)

function for producer initialization.

Parameters

- **device** (*str*) – device for the this process.
- **cfg** (easycore.common.config.CfgNode) – config of this process, you can use it to transfer data to *producer_work* and *producer_end* function.

static producer_work(*device*, *cfg*, *data*)

function specify how the producer processes the data.

Parameters

- **device** (*str*) – device for this process.
- **cfg** (easycore.common.config.CfgNode) – config of this process, you can use it to get data from *producer_init* function and transfer data to the next *producer_work* and *producer_end* function.
- **data** (*Any*) – data get from input of *__call__* method.

Returns *Any* – processed data

static producer_end(*device*, *cfg*)

function after finishing all of its task and before close the process.

Parameters

- **device** (*str*) – device for this process.
- **cfg** (easycore.common.config.CfgNode) – config of this process, you can use it to get data from *producer_init* and *producer_work* function.

static consumer_init(*cfg*)

function for consumer initialization.

Parameters **cfg** (easycore.common.config.CfgNode) – config of this process, you can use it to transfer data to *consumer_work* and *consumer_end* function.

static consumer_work(*cfg*, *data*)

function specify how the consumer processses the data from producers.

Parameters **cfg** (easycore.common.config.CfgNode) – config of this process, you can use it to get data from *consumer_init* function and transfer data to the next *consumer_work* and *consumer_end* function.

static consumer_end(*cfg*)
function after receiving all data from producers.

Parameters **cfg** (easycore.common.config.CfgNode) – config of this process, you can use it get data from *consumer_work* function.

Returns *Any* – processed data

__call__(*data_iter*)

Parameters **data_iter** (*Iterable*) – iterator of data

Returns *Any* – result

close()
Shutdown all processes if this runner is alive.

activate()
Restart all processes if this runner is closed.

class easycore.common.parallel.**UnorderedRunner**(*devices*, *cfg={}*, *queue_scale=3.0*)
Bases: easycore.common.parallel.engine.BaseRunner

A Multi-process runner whose consumer receive data in unorder. The runner will start multi-processes for producers and 1 thread for consumer.

__init__(*devices*, *cfg={}*, *queue_scale=3.0*)

Parameters

- **devices** (*int or Iterable*) – If the *devices* is *int*, it will use devices cpu to do the work. If the *devices* is an iterable object, such as list, it will use the devices specified by the iterable object, such as ["cpu", "cuda:0", "cuda:1"].
- **cfg** (easycore.common.config.CfgNode) – user custom data.
- **queue_scale** (*float*) – scale the queues for communication between processes.

__call__(*data_iter*)

Parameters **data_iter** (*Iterable*) – iterator of data

Returns *Any* – result

activate()
Restart all processes if this runner is closed.

close()
Shutdown all processes if this runner is alive.

static consumer_end(*cfg*)
function after receiving all data from producers.

Parameters **cfg** (easycore.common.config.CfgNode) – config of this process, you can use it get data from *consumer_work* function.

Returns *Any* – processed data

static consumer_init(*cfg*)
function for consumer initialization.

Parameters **cfg** (easycore.common.config.CfgNode) – config of this process, you can use it to transfer data to *consumer_work* and *consumer_end* function.

static consumer_work(*cfg*, *data*)
function specify how the consumer processses the data from producers.

Parameters **cfg** (easycore.common.config.CfgNode) – config of this process, you can use it to get data from *consumer_init* function and transfer data to the next *consumer_work* and *consumer_end* function.

is_activate
whether the runner is alive.

static producer_end(*device*, *cfg*)
function after finishing all of its task and before close the process.

Parameters

- **device** (*str*) – device for this process.
- **cfg** (easycore.common.config.CfgNode) – config of this process, you can use it to get data from *producer_init* and *producer_work* function.

static producer_init(*device*, *cfg*)
function for producer initialization.

Parameters

- **device** (*str*) – device for the this process.
- **cfg** (easycore.common.config.CfgNode) – config of this process, you can use it to transfer data to *producer_work* and *producer_end* function.

static producer_work(*device*, *cfg*, *data*)
function specify how the producer processes the data.

Parameters

- **device** (*str*) – device for this process.
- **cfg** (easycore.common.config.CfgNode) – config of this process, you can use it to get data from *producer_init* function and transfer data to the next *producer_work* and *producer_end* function.
- **data** (*Any*) – data get from input of *__call__* method.

Returns *Any* – processed data

class easycore.common.parallel.**OrderedRunner**(*devices*, *cfg={}*, *queue_scale=3.0*)
Bases: easycore.common.parallel.engine.BaseRunner

A Multi-process runner whose consumer receive data in order. The runner will start multi-processes for producers and 1 thread for consumer.

__init__(*devices*, *cfg={}*, *queue_scale=3.0*)

Parameters

- **devices** (*int or Iterable*) – If the *devices* is *int*, it will use devices cpu to do the work. If the *devices* is an iterable object, such as list, it will use the devices specified by the iterable object, such as ["cpu", "cuda:0", "cuda:1"].
- **cfg** (easycore.common.config.CfgNode) – user custom data.
- **queue_scale** (*float*) – scale the queues for communication between processes.

close()
Shutdown all processes if this runner is alive.

activate()
Restart all processes if this runner is closed.

__call__(*data_iter*)

Parameters **data_iter** (*Iterable*) – iterator of data

Returns *Any* – result

static consumer_end(*cfg*)
function after receiving all data from producers.

Parameters **cfg** (easycore.common.config.CfgNode) – config of this process, you can use it get data from *consumer_work* function.

Returns *Any* – processed data

static consumer_init(*cfg*)
function for consumer initialization.

Parameters **cfg** (easycore.common.config.CfgNode) – config of this process, you can use it to transfer data to *consumer_work* and *consumer_end* function.

static consumer_work(*cfg*, *data*)
function specify how the consumer processses the data from producers.

Parameters **cfg** (easycore.common.config.CfgNode) – config of this process, you can use it to get data from *consumer_init* function and transfer data to the next *consumer_work* and *consumer_end* function.

is_activate
whether the runner is alive.

static producer_end(*device*, *cfg*)
function after finishing all of its task and before close the process.

Parameters

- **device** (*str*) – device for this process.
- **cfg** (easycore.common.config.CfgNode) – config of this process, you can use it to get data from *producer_init* and *producer_work* function.

static producer_init(*device*, *cfg*)
function for producer initialization.

Parameters

- **device** (*str*) – device for the this process.
- **cfg** (easycore.common.config.CfgNode) – config of this process, you can use it to transfer data to *producer_work* and *producer_end* function.

static producer_work(*device*, *cfg*, *data*)
function specify how the producer processes the data.

Parameters

- **device** (*str*) – device for this process.
- **cfg** (easycore.common.config.CfgNode) – config of this process, you can use it to get data from *producer_init* function and transfer data to the next *producer_work* and *producer_end* function.
- **data** (*Any*) – data get from input of *__call__* method.

Returns *Any* – processed data

2.1.3 easycore.common.registry

class easycore.common.registry.**Registry**(*name: str*)

Bases: object

The registry that provides name -> object mapping.

To create a registry:

```
MODEL_REGISTRY = Registry("MODEL")
```

To register an object with its __name__:

```
@MODEL_REGISTRY.register()
class ResNet50:
    pass

# or

MODEL_REGISTRY.register(obj=ResNet50)
```

To register an object with a given name:

```
@MODEL_REGISTRY.register("resnet")
class RestNet50:
    pass

# or

MODEL_REGISTRY.register("resnet", ResNet50)
```

To get a registered object from registry:

```
model_class = MODEL_REGISTRY.get("ResNet50")

# or

model_class = MODEL_REGISTRY.get("resnet")
```

__init__(*name: str*) → None

Parameters **name** (*str*) – name of this registry

register(*name: str = None, obj: object = None*) → Optional[object]

Register the given object with given name. If the object is not given, it will act as a decorator.

Parameters

- **name** (*str, optional*) – if not given, it will use *obj.__name__* as the name.
- **obj** (*object, optional*) – if not given, this method will return a decorator.

Returns *Optional[object]* – None or a decorator.

unregister(*name: str*) → None

Remove registered object.

Parameters **name** (*str*) – registered name

is_registered(*name*)

Get whether the given name has been registered.

Parameters **name** (*str*) –

Returns *bool* – whether the name has been registered.

get (*name: str*) → object
Get a registered object from registry by its name.

Parameters **name** (*str*) – registered name.

Returns *object* – registered object.

registered_names () → List[str]
Get all registered names.

Returns *list[str]* – list of registered names.

2.2 easycore.torch

2.2.1 easycore.torch.parallel

class easycore.torch.parallel.**BaseRunner** (*devices*, *cfg={}*, *queue_scale=3.0*)
Bases: object

A Multi-process runner whose consumer receive data in unorder. The runner will start multi-processes for producers and 1 thread for consumer.

__init__ (*devices*, *cfg={}*, *queue_scale=3.0*)

Parameters

- **devices** (*int or Iterable*) – If the *devices* is *int*, it will use devices cpu to do the work. If the *devices* is an iterable object, such as list, it will use the devices specified by the iterable object, such as ["cpu", "cuda:0", "cuda:1"].
- **cfg** (easycore.common.config.CfgNode) – user custom data.
- **queue_scale** (*float*) – scale the queues for communication between processes.

is_activate
whether the runner is alive.

static producer_init (*device*, *cfg*)
function for producer initialization.

Parameters

- **device** (*str*) – device for the this process.
- **cfg** (easycore.common.config.CfgNode) – config of this process, you can use it to transfer data to *producer_work* and *producer_end* function.

static producer_work (*device*, *cfg*, *data*)
function specify how the producer processes the data.

Parameters

- **device** (*str*) – device for this process.
- **cfg** (easycore.common.config.CfgNode) – config of this process, you can use it to get data from *producer_init* function and transfer data to the next *producer_work* and *producer_end* function.
- **data** (*Any*) – data get from input of *__call__* method.

Returns *Any* – processed data

static producer_end(*device*, *cfg*)
function after finishing all of its task and before close the process.

Parameters

- **device** (*str*) – device for this process.
- **cfg** (easycore.common.config.CfgNode) – config of this process, you can use it to get data from *producer_init* and *producer_work* function.

static consumer_init(*cfg*)
function for consumer initialization.

Parameters **cfg** (easycore.common.config.CfgNode) – config of this process, you can use it to transfer data to *consumer_work* and *consumer_end* function.

static consumer_work(*cfg*, *data*)
function specify how the consumer processses the data from producers.

Parameters **cfg** (easycore.common.config.CfgNode) – config of this process, you can use it to get data from *consumer_init* function and transfer data to the next *consumer_work* and *consumer_end* function.

static consumer_end(*cfg*)
function after receiving all data from producers.

Parameters **cfg** (easycore.common.config.CfgNode) – config of this process, you can use it get data from *consumer_work* function.

Returns *Any* – processed data

__call__(*data_iter*)

Parameters **data_iter** (*Iterable*) – iterator of data

Returns *Any* – result

close()
Shutdown all processes if this runner is alive.

activate()
Restart all processes if this runner is closed.

class easycore.torch.parallel.**UnorderedRunner**(*devices*, *cfg={}*, *queue_scale=3.0*)
Bases: easycore.torch.parallel.engine.BaseRunner

A Multi-process runner whose consumer receive data in unorder. The runner will start multi-processes for producers and 1 thread for consumer.

__init__(*devices*, *cfg={}*, *queue_scale=3.0*)

Parameters

- **devices** (*int or Iterable*) – If the *devices* is *int*, it will use devices cpu to do the work. If the *devices* is an iterable object, such as list, it will use the devices specified by the iterable object, such as ["cpu", "cuda:0", "cuda:1"].
- **cfg** (easycore.common.config.CfgNode) – user custom data.
- **queue_scale** (*float*) – scale the queues for communication between processes.

__call__(*data_iter*)

Parameters **data_iter** (*Iterable*) – iterator of data

Returns *Any* – result

activate()
Restart all processes if this runner is closed.

close()
Shutdown all processes if this runner is alive.

static consumer_end(*cfg*)
function after receiving all data from producers.

Parameters **cfg** (easycore.common.config.CfgNode) – config of this process, you can use it get data from *consumer_work* function.

Returns *Any* – processed data

static consumer_init(*cfg*)
function for consumer initialization.

Parameters **cfg** (easycore.common.config.CfgNode) – config of this process, you can use it to transfer data to *consumer_work* and *consumer_end* function.

static consumer_work(*cfg*, *data*)
function specify how the consumer processses the data from producers.

Parameters **cfg** (easycore.common.config.CfgNode) – config of this process, you can use it to get data from *consumer_init* function and transfer data to the next *consumer_work* and *consumer_end* function.

is_activate
whether the runner is alive.

static producer_end(*device*, *cfg*)
function after finishing all of its task and before close the process.

Parameters

- **device** (*str*) – device for this process.
- **cfg** (easycore.common.config.CfgNode) – config of this process, you can use it to get data from *producer_init* and *producer_work* function.

static producer_init(*device*, *cfg*)
function for producer initialization.

Parameters

- **device** (*str*) – device for the this process.
- **cfg** (easycore.common.config.CfgNode) – config of this process, you can use it to transfer data to *producer_work* and *producer_end* function.

static producer_work(*device*, *cfg*, *data*)
function specify how the producer processes the data.

Parameters

- **device** (*str*) – device for this process.
- **cfg** (easycore.common.config.CfgNode) – config of this process, you can use it to get data from *producer_init* function and transfer data to the next *producer_work* and *producer_end* function.
- **data** (*Any*) – data get from input of *__call__* method.

Returns *Any* – processed data

class easycore.torch.parallel.**OrderedRunner**(*devices*, *cfg={}*, *queue_scale=3.0*)

Bases: easycore.torch.parallel.engine.BaseRunner

A Multi-process runner whose consumer receive data in order. The runner will start multi-processes for producers and 1 thread for consumer.

__init__(*devices*, *cfg={}*, *queue_scale=3.0*)

Parameters

- **devices** (*int or Iterable*) – If the *devices* is *int*, it will use devices cpu to do the work. If the *devices* is an iterable object, such as list, it will use the devices specified by the iterable object, such as ["cpu", "cuda:0", "cuda:1"].
- **cfg** (easycore.common.config.CfgNode) – user custom data.
- **queue_scale** (*float*) – scale the queues for communication between processes.

close()

Shutdown all processes if this runner is alive.

activate()

Restart all processes if this runner is closed.

__call__(*data_iter*)

Parameters **data_iter** (*Iterable*) – iterator of data

Returns *Any* – result

static consumer_end(*cfg*)

function after receiving all data from producers.

Parameters **cfg** (easycore.common.config.CfgNode) – config of this process, you can use it get data from *consumer_work* function.

Returns *Any* – processed data

static consumer_init(*cfg*)

function for consumer initialization.

Parameters **cfg** (easycore.common.config.CfgNode) – config of this process, you can use it to transfer data to *consumer_work* and *consumer_end* function.

static consumer_work(*cfg*, *data*)

function specify how the consumer processses the data from producers.

Parameters **cfg** (easycore.common.config.CfgNode) – config of this process, you can use it to get data from *consumer_init* function and transfer data to the next *consumer_work* and *consumer_end* function.

is_activate

whether the runner is alive.

static producer_end(*device*, *cfg*)

function after finishing all of its task and before close the process.

Parameters

- **device** (*str*) – device for this process.
- **cfg** (easycore.common.config.CfgNode) – config of this process, you can use it to get data from *producer_init* and *producer_work* function.

static producer_init(*device*, *cfg*)

function for producer initialization.

Parameters

- **device** (*str*) – device for the this process.
- **cfg** (easycore.common.config.CfgNode) – config of this process, you can use it to transfer data to *producer_work* and *producer_end* function.

static producer_work(*device*, *cfg*, *data*)

function specify how the producer processes the data.

Parameters

- **device** (*str*) – device for this process.
- **cfg** (easycore.common.config.CfgNode) – config of this process, you can use it to get data from *producer_init* function and transfer data to the next *producer_work* and *producer_end* function.
- **data** (*Any*) – data get from input of *__call__* method.

Returns *Any* – processed data

CHAPTER 3

Indices and tables

- genindex
- search

Python Module Index

e

Index

Symbols

A

B

C

D

E

F

G

H

I

L

M

O

P

R

S

U