Xmodaler Documentation

Release 1.1.0

Yehao Li

Feb 27, 2023

CONTENTS

X-modaler is a versatile and high-performance codebase for cross-modal analytics. This codebase unifies comprehensive high-quality modules in state-of-the-art vision-language techniques, which are organized in a standardized and user-friendly fashion.

CHAPTER

ONE

TUTORIALS

1.1 Installation

1.1.1 Requiremenets

- Linux or macOS with Python $\geqslant$ 3.6
- PyTorch $\geqslant$ 1.8 and torchvision that matches the PyTorch installation. Install them together at pytorch.org to make sure of this
- fvcore
- pytorch_transformers
- jsonlines
- pycocotools
- omegaconf
- cloudpickle
- tensorboard

1.1.2 Download X-modaler

After satisfying all the requirements, run:

```
git clone https://github.com/YehLi/xmodaler.git --recursive
```

Now, you can build your own frameworks using APIs supported by X-modaler !

1.2 Getting Started with X-modaler

This document provides a brief intro of the usage of builtin command-line tools in X-modaler, showing how to run inference with an existing model, and how to train a builtin model on a custom dataset (e.g., MSCOCO or MSVD).

1.2.1 Training & Evaluation in Command Line

We provide a script in "train_net.py", which is made to train all the configs provided in X-modaler. You may want to use it as a reference to write your own training script.

To train a model(e.g., UpDown) with "train_net.py", first setup the corresponding datasets following datasets, then run:

```
# Teacher Force
python train_net.py --num-gpus 4 \
         --config-file configs/image_caption/updown.yaml

# Reinforcement Learning
python train_net.py --num-gpus 4 \
         --config-file configs/image_caption/updown_rl.yaml
```

1.3 Use Builtin Datasets

A dataset can be used by wrapping it into a torch Dataset. This document explains how to setup the builtin datasets so they can be used by X-modaler. The annotations for builtin datasets can be downloaded here.

X-modaler has builtin supports for a few datasets (e.g., MSCOCO or MSVD). The corresponding dataset wrappers are provided in `xmodaler/datasets`:

```
xmodaler/datasets/
  images/
    mscoco.py
  videos/
    msvd.py
```

You can specify which dataset wrapper to use by `DATASETS.TRAIN`, `DATASETS.VAL` and `DATASETS.TEST` in the config file.

1.3.1 Expected dataset structure for COCO:

```
mscoco_dataset/
  mscoco_caption_anno_train.pkl
  mscoco_caption_anno_val.pkl
  mscoco_caption_anno_test.pkl
  vocabulary.txt
  captions_val5k.json
  captions_test5k.json
  # image files that are mentioned in the corresponding json
  features/
    up_down/
      *.npz
```

1.3.2 Expected dataset structure for MSVD:

```
msvd_dataset/
  msvd_caption_anno_train.pkl
  msvd_caption_anno_val.pkl
  msvd_caption_anno_test.pkl
  vocabulary.txt
  captions_val.json
  captions_test.json
  # videos files that are mentioned in the corresponding json
  features/
    resnet152/
      *.npy
```

1.3.3 Expected dataset structure for MSR-VTT:

```
msrvtt_dataset/
  msrvtt_caption_anno_train.pkl
  msrvtt_caption_anno_val.pkl
  msrvtt_caption_anno_test.pkl
  vocabulary.txt
  captions_val.json
  captions_test.json
  # videos files that are mentioned in the corresponding json
  msrvtt_torch/
    feature/
          resnet152/
            *.npy
```

When the dataset wrapper and data files are ready, you need to specify the corresponding paths to these data files in the config file. For example,

```
DATALOADER:
        FEATS_FOLDER: '../open_source_dataset/mscoco_dataset/features/up_down'    #␣
↪feature folder
        ANNO_FOLDER: '../open_source_dataset/mscoco_dataset' # annotation folders
INFERENCE:
        VOCAB: '../open_source_dataset/mscoco_dataset/vocabulary.txt'
        VAL_ANNFILE: '../open_source_dataset/mscoco_dataset/captions_val5k.json'
        TEST_ANNFILE:  '../open_source_dataset/mscoco_dataset/captions_test5k.json'
```

1.4 Extend X-modaler's Defaults

X-modaler provides very thin abstractions in APIs to allow for the possibility of doing everything in new ways, which means that it is easy to break existing abstractions and replace them with new ones. Also, high-level abstractions are available in X-modaler so that users can easily do things in standard ways, without worrying about the details that only certain researchers care about.

In X-modaler, the following interfaces are supported:

- Functions and classes that take a config (**cfg**) argument (sometimes with few extra arguments) as input. Such functions and classes implement the **"standard default"** behavior: it will read what it needs from the config. Users only need to load a given config and pass it around.
- Functions and classes that have well-defined explicit arguments, which requires users' expertise to understand what each argument should be.
- A few functions and classes are implemented with the **@configurable** decorator - they can be called with either a config, or with explicit arguments.

As an example, a UpDown model can be built in the following ways:

```
cfg = ... # read the config
model = build_model(cfg) # users only need to pass a config
```

If you only need the standard behavior, the Beginner's Tutorial should suffice. If you need to extend X-modaler to your own needs, see the following tutorials for more details:

- X-modaler includes a few standard datasets. To use custom ones, see Use Custom Datasets.
- X-modaler contains the standard logic that creates a data loader for training/testing from a dataset, but you can write your own as well. See DataLoader.
- X-modaler implements several state-of-the-art models for visoin-and-language tasks, and provides ways for you to overwrite their behaviors. See Use Models and Write Models.
- X-modaler provides a default training loop that is good for common training tasks. You can customize it with hooks, or write your own loop instead. See *Training*.

1.5 Use Custom Datasets

Datasets that have builtin support in X-modaler are listed in builtin datasets. If you want to use a custom dataset while also reusing X-modaler's data loaders, you will need to **register** your dataset (i.e., tell X-modaler how to obtain your dataset).

1.5.1 Register a Dataset

To let X-modaler know how to obtain a dataset named "MyDataset", users need to implement a class either in the folder `xmodaler/datasets/images` or `xmodaler/datasets/videos` that returns the item in your dataset and then tell X-modaler about this class:

```
from ..build import DATASETS_REGISTRY

__all__ = ["MyDataset"]

@DATASETS_REGISTRY.register()
class MyDataset:
    @configurable
    def __init__(
        self,
        stage: str,
        anno_file: str,
        seq_per_img: int,
        max_feat_num: int,
```

(continues on next page)

(continued from previous page)

```
        max_seq_len: int,
        feats_folder: str
    ):
        # load dataset and set hyperparameters
        ...

    @classmethod
    def from_config(cls, cfg, stage: str = "train"):
        ...
        return arguments

    def __call__(self, dataset_dict):
        # process a data item for a specific task
        ...
        return item_dict
```

The function can do arbitrary things and should return the data dict in either of the following formats:

- X-modaler's standard dataset dict using builtin keys defined by `xmodaler.config.kfg`. This will make it work with many other builtin features in X-modaler, so it's recommended to use it when it's sufficient.
- Any custom format. You can also return arbitrary dict in your own format by adding extra keys to `xmodaler.config.kfg` for new tasks. Then you will need to handle them properly downstream as well.

1.6 Dataloader

Dataloader is the component that provides data to models. A dataloader usually (but not necessarily) takes raw information from datasets, and process them into a format needed by the model.

1.6.1 How the Existing Dataloader Works

X-modaler contains a builtin data loading pipeline. It's good to understand how it works, in case you need to write a custom one.

X-modaler provides a function `xmodaler.datasets.build_xmodaler_{train,valtest}_loader` that creates a default dataloader from a given config. Here is how `build_xmodaler_{train,valtest}_loader` works:

- It takes a helper class (e.g., `xmodaler.datasets.common.DatasetFromList`) and loads a **list[dict]** representing the dataset items in a lightweight format. These dataset items are not yet ready to be used by the model (e.g., images are not loaded into memory).
- Each dict in this list is processed by the class `xmodaler.datasets.common.MapDataset`. Users can customize this data loading for specific datasets by implementing the `__call__` function in a wapper class (e.g., `xmodaler.datasets.MSCoCoDataset`), which is one of the arguments to initialize `MapDataset`. The role of the wapper class is to transform the lightweight representation of a dataset item into a format that is ready for the model to consume (including, e.g., read images, caption sampling or convert to torch Tensors).
- After gathering a list of items, batching schema is handled by defining the argument `collate_fn` of `torch.utils.data.DataLoader` in `xmodaler.datasets.build_xmodaler_{train,valtest}_loader` functions.

The batched data is the output of the data loader. Typically, it's also the input of `model.forward()`.

1.6.2 Write a Custom Dataloader

Using a different "wapper class" as the argument `dataset_mapper` with `build_xmodaler_{train, valtest}_loader` works for most cases of custom data loading. See Use Custom Datasets to custom the "wapper class".

1.6.3 Use a Custom Dataloader

If you use *DefaultTrainer*, you can overwrite its `build_xmodaler_{train,valtest}_loader` method to use your own dataloader.

1.7 Use Models

Models (and their sub-models) in X-modaler are built by the function `build_model`:

```
from xmodaler.modeling import build_model
cfg = ... # read the config
model = build_model(cfg)
```

`build_model` only builds the model structure and fills it with random parameters defined in the given config (**cfg**). See below for how to load an existing checkpoint to the model and how to use the model object.

1.7.1 Load/Save a Checkpoint

Users need to specify the path of the checkpoint to load by setting the argument `MODEL.WEIGHTS` in the given config (**cfg**):

```
from xmodaler.checkpoint import XmodalerCheckpointer
checkpointer = XmodalerCheckpointer(model, cfg.OUTPUT_DIR)
checkpointer.resume_or_load(cfg.MODEL.WEIGHTS, resume=True) # load a checkpoint
checkpointer.save("model_00") # save model
```

1.7.2 Use a Model

A model can be called by `outputs = model(inputs)`, where inputs is the batched output of dataloader. Each item in the dict `inputs` corresponds to a batch of images/videos and the required inputs depend on the type of model and task.

Training: When in training mode, all models are required to be used under an EventStorage. The training statistics will be put into the storage:

```
with EventStorage(start_iter) as self.storage:
    outputs_dict = model(inputs)
    losses = loss(outputs_dict)
```

You can run inference directly like this:

```
model.eval()
with torch.no_grad():
    outputs = model(inputs)
```

1.8 Write Models

If you are trying to do something completely new, you may wish to implement a model entirely from scratch. However, in many situations you may be interested in modifying or extending some components of an existing model. Therefore, we also provide mechanisms that let users override the behavior of certain internal components of standard models.

1.8.1 Register New Components

For common concepts that users often want to customize, such as "encoder/decoder", "layers", we provide a registration mechanism for users to inject custom implementations that will be immediately available to use in config files.

For example, to add a new encoder to `xmodaler/modeling/encoder`, import this code in your code:

```
from xmodaler.modeling.encoder.build import ENCODER_REGISTRY

__all__ = ["MyEncoder"]

@ENCODER_REGISTRY.register()
class MyEncoder(nn.Module):
    @configurable
    def __init__(self):
        super(MyEncoder, self).__init__()

    @classmethod
    def from_config(cls, cfg):
        ...
        return {} # return init arugments

    @classmethod
    def add_config(cls, cfg):
        ...

    def forward(self, batched_inputs, mode=None):
        ...
        return outputs
```

In this code, we implement a new encoder following the interface of the `torch.nn.Module`, and register it into the `ENCODER_REGISTRY`. After importing this code, X-modaler can link the name of the class to its implementation. Therefore you can write the following code:

```
cfg = ...   # read a config
cfg.MODEL.ENCODER = 'MyEncoder' # or set it in the config file
model = build_model(cfg) # it will find `MyEncoder` defined above
```

1.9 Training

From the previous tutorials, you may now have a custom model and a dataloader. To run training, users typically have a preference in the following style:

1.9.1 Trainer Abstraction

X-modaler provides a standarized "trainer" abstraction with a hook system that helps simplify the standard training behavior. It includes the following instantiation:

- `DefaultTrainer` is a default trainer initialized from a config, used by many scripts. It includes more standard default behaviors that one might want to opt in, including default configurations for optimizer, learning rate scheduling, logging, evaluation, checkpointing etc.

Users can write their custom trainer by following the class `DefaultTrainer`.

1.10 Evaluation

Evaluation is a process that takes a number of inputs/outputs pairs and aggregate them. You can always use the model directly and just parse its inputs/outputs manually to perform evaluation. Alternatively, evaluation is implemented in `xmodaler.evaluation`.

1.10.1 Custom Evaluators

X-modaler includes a few evaluators that compute metrics using standard dataset-specific APIs (e.g., COCO Captions). You can also implement your own evaluator that performs some other jobs using the inputs/outputs pairs:

```
from xmodaler.evaluation.build import EVALUATION_REGISTRY
@EVALUATION_REGISTRY.register()
class MyEvaler(object):
    def __init__(self, cfg, annfile):
        super(MyEvaler, self).__init__()
        ...

    def eval(self, result):
        ...
```

1.10.2 Use Evaluators

To evaluate using the methods of evaluators manually:

```
evaluator = MyEvaler(cfg, reference_file)
model.eval()
with torch.no_grad():
    outputs = model(inputs)
eval_results = evaluator.eval(outputs)
```

1.11 Configs

X-modaler provides a key-value based config system that can be used to obtain standard, common behaviors.

X-modaler's config system uses YAML. In addition to the basic operations that access and update a config, we provide the following extra functionality:

- The config can have `_BASE_: base.yaml` field, which will load a base config first. Values in the base config will be overwritten in sub-configs, if there are any conflicts. We provided several base configs for standard model architectures in the folder `configs/`.

1.11.1 Basic Usage

Some basic usage of the `CfgNode` object is shown here:

```
from xmodaler.config import get_cfg
from xmodaler.modeling import add_config
cfg = get_cfg() # obtain X-modaler's default config
tmp_cfg = cfg.load_from_file_tmp(config_file) # load custom config
add_config(cfg, tmp_cfg) # combining default and custom configs
cfg.merge_from_file(config_file) # load values from a file
cfg.merge_from_list(opts) # # can also load values from a list of str
```

Many builtin tools in X-modaler accept command line config overwrite: key-value pairs provided in the command line will overwrite the existing values in the config file. For example,

```
python train_net.py --config-file config.yaml [--other-options] \
  --opts MODEL.WEIGHTS /path/to/weights
```

CHAPTER

TWO

NOTES

2.1 Benchmarks

2.1.1 Image Captioning on MSCOCO (Cross-Entropy Loss)

Model	BLEU@1	BLEU@2	BLEU@3	BLEU@4	METEOR	ROUGE-L	CIDEr-D	SPICE
LSTM-A3	75.3	59.0	45.4	35.0	26.7	55.6	107.7	19.7
Attention	76.4	60.6	46.9	36.1	27.6	56.6	113.0	20.4
Up-Down	76.3	60.3	46.6	36.0	27.6	56.6	113.1	20.7
GCN-LSTM	76.8	61.1	47.6	36.9	28.2	57.2	116.3	21.2
Transformer	76.4	60.3	46.5	35.8	28.2	56.7	116.6	21.3
Meshed-Memory Transformer	76.3	60.2	46.4	35.6	28.1	56.5	116.0	21.2
X-LAN	77.5	61.9	48.3	37.5	28.6	57.6	120.7	21.9
TDEN	75.5	59.4	45.7	34.9	28.7	56.7	116.3	22.0

2.1.2 Image Captioning on MSCOCO (CIDEr Score Optimization)

Model	BLEU@1	BLEU@2	BLEU@3	BLEU@4	METEOR	ROUGE-L	CIDEr-D	SPICE
LSTM-A3	77.9	61.5	46.7	35.0	27.1	56.3	117.0	20.5
Attention	79.4	63.5	48.9	37.1	27.9	57.6	123.1	21.3
Up-Down	80.1	64.3	49.7	37.7	28.0	58.0	124.7	21.5
GCN-LSTM	80.2	64.7	50.3	38.5	28.5	58.4	127.2	22.1
Transformer	80.5	65.4	51.1	39.2	29.1	58.7	130.0	23.0
Meshed-Memory Transformer	80.7	65.5	51.4	39.6	29.2	58.9	131.1	22.9
X-LAN	80.4	65.2	51.0	39.2	29.4	59.0	131.0	23.2
TDEN	81.3	66.3	52.0	40.1	29.6	59.8	132.6	23.4

2.1.3 Video Captioning on MSVD

Model	BLEU@1	BLEU@2	BLEU@3	BLEU@4	METEOR	ROUGE-L	CIDEr-D	SPICE
MP-LSTM	77.0	65.6	56.9	48.1	32.4	68.1	73.1	4.8
TA	80.4	68.9	60.1	51.0	33.5	70.0	77.2	4.9
Transformer	79.0	67.6	58.5	49.4	33.3	68.7	80.3	4.9
TD-ConvED	81.6	70.4	61.3	51.7	34.1	70.4	77.8	5.0

2.1.4 Video Captioning on MSR-VTT

Model	BLEU@1	BLEU@2	BLEU@3	BLEU@4	METEOR	ROUGE-L	CIDEr-D	SPICE
MP-LSTM	73.6	60.8	49.0	38.6	26.0	58.3	41.1	5.6
TA	74.3	61.8	50.3	39.9	26.4	59.4	42.9	5.8
Transformer	75.4	62.3	50.0	39.2	26.5	58.7	44.0	5.9
TD-ConvED	76.4	62.3	49.9	38.9	26.3	59.0	40.7	5.7

2.1.5 Visual Question Answering

Model	Overall	Yes/No	Number	Other
Uniter	70.1	86.8	53.7	59.6
TDEN	71.9	88.3	54.3	62.0

2.1.6 Caption-based image retrieval on Flickr30k

Model	R1	R5	R10
Uniter	61.6	87.7	92.8
TDEN	62.0	86.6	92.4

2.1.7 Visual commonsense reasoning

Model	Q -> A	QA -> R	Q -> AR
Uniter	73.0	75.3	55.4
TDEN	75.0	76.5	57.7

CHAPTER

THREE

API DOCUMENTATION

3.1 xmodaler.checkpoint

class xmodaler.checkpoint.**PeriodicEpochCheckpointer**(*checkpointer: Checkpointer*, *period: int*, *max_iter: Optional[int] = None*, *max_to_keep: Optional[int] = None*, *file_prefix: str = 'model'*)

Bases: PeriodicCheckpointer

__init__(*checkpointer: Checkpointer*, *period: int*, *max_iter: Optional[int] = None*, *max_to_keep: Optional[int] = None*, *file_prefix: str = 'model'*) → None

Parameters

- **checkpointer** – the checkpointer object used to save checkpoints.
- **period** (*int*) – the period to save checkpoint.
- **max_iter** (*int*) – maximum number of iterations. When it is reached, a checkpoint named "{file_prefix}_final" will be saved.
- **max_to_keep** (*int*) – maximum number of most current checkpoints to keep, previous checkpoints will be deleted
- **file_prefix** (*str*) – the prefix of checkpoint's filename

save(*name: str*, ***kwargs: Any*) → None

Same argument as `Checkpointer.save()`. Use this method to manually save checkpoints outside the schedule.

Parameters

- **name** (*str*) – file name.
- **kwargs** (*Any*) – extra data to save, same as in `Checkpointer.save()`.

step(*iteration: int*, *epoch: int*, ***kwargs: Any*) → None

Perform the appropriate action at the given iteration.

Parameters

- **iteration** (*int*) – the current iteration, ranged in [0, max_iter-1].
- **kwargs** (*Any*) – extra data to save, same as in `Checkpointer.save()`.

class xmodaler.checkpoint.**XmodalerCheckpointer**(*model*, *save_dir=''*, ***, *save_to_disk=None*, ***checkpointables*)

Bases: `Checkpointer`

Same as `Checkpointer`, but is able to handle models in xmodaler model zoo, and apply conversions for legacy models.

__init__(*model*, *save_dir=''*, *, *save_to_disk=None*, ***checkpointables*)

Parameters

- **model** (*nn.Module*) – model.
- **save_dir** (*str*) – a directory to save and find checkpoints.
- **save_to_disk** (*bool*) – if True, save checkpoint to disk, otherwise disable saving for this checkpointer.
- **checkpointables** (*object*) – any checkpointable objects, i.e., objects that have the `state_dict()` and `load_state_dict()` method. For example, it can be used like *Checkpointer(model, "dir", optimizer=optimizer)*.

_convert_ndarray_to_tensor(*state_dict: Dict[str, Any]*) → None

In-place convert all numpy arrays in the state_dict to torch tensor. :param state_dict: a state-dict to be loaded to the model.

Will be modified.

_load_file(*filename*)

Load a checkpoint file. Can be overwritten by subclasses to support different formats.

Parameters

f (*str*) – a locally mounted file path.

Returns

with keys "model" and optionally others that are saved by
the checkpointer dict["model"] must be a dict which maps strings to torch.Tensor or numpy arrays.

Return type
dict

_load_model(*checkpoint*)

Load weights from a checkpoint.

Parameters

checkpoint (*Any*) – checkpoint contains the weights.

Returns

`NamedTuple` with `missing_keys`, `unexpected_keys`,
and `incorrect_shapes` fields: * **missing_keys** is a list of str containing the missing keys * **unexpected_keys** is a list of str containing the unexpected keys * **incorrect_shapes** is a list of (key, shape in checkpoint, shape in model)

This is just like the return value of `torch.nn.Module.load_state_dict()`, but with extra support for `incorrect_shapes`.

_log_incompatible_keys(*incompatible: _IncompatibleKeys*) → None

Log information about the incompatible keys returned by `_load_model`.

add_checkpointable(*key: str*, *checkpointable: Any*) → None

Add checkpointable object for this checkpointer to track.

Parameters

- **key** (*str*) – the key used to save the object
- **checkpointable** – any object with `state_dict()` and `load_state_dict()` method

get_all_checkpoint_files() → List[str]

Returns

All available checkpoint files (.pth files) in target
directory.

Return type
list

get_checkpoint_file() → str

Returns
The latest checkpoint file in target directory.

Return type
str

has_checkpoint() → bool

Returns
whether a checkpoint exists in the target directory.

Return type
bool

load(*path: str*, *checkpointables: Optional[List[str]] = None*) → Dict[str, Any]

Load from the given checkpoint.

Parameters

- **path** (*str*) – path or url to the checkpoint. If empty, will not load anything.
- **checkpointables** (*list*) – List of checkpointable names to load. If not specified (None), will load all the possible checkpointables.

Returns
extra data loaded from the checkpoint that has not been processed. For example, those saved with `save(**extra_data)()`.

Return type
dict

resume_or_load(*path: str*, *, *resume: bool = True*) → Dict[str, Any]

If *resume* is True, this method attempts to resume from the last checkpoint, if exists. Otherwise, load checkpoint from the given path. This is useful when restarting an interrupted training job.

Parameters

- **path** (*str*) – path to the checkpoint.
- **resume** (*bool*) – if True, resume from the last checkpoint if it exists and load the model together with all the checkpointables. Otherwise only load the model without loading any checkpointables.

Returns
same as `load()`.

save(*name: str*, ***kwargs: Any*) → None

Dump model and checkpointables to a file.

Parameters

- **name** (*str*) – name of the file.
- **kwargs** (*dict*) – extra arbitrary data to save.

tag_last_checkpoint(*last_filename_basename: str*) → None

Tag the last checkpoint.

Parameters
last_filename_basename (*str*) – the basename of the last filename.

3.2 xmodaler.config

class xmodaler.config.**CfgNode**(*init_dict=None*, *key_list=None*, *new_allowed=False*)

Bases: CfgNode

The same as *fvcore.common.config.CfgNode*, but different in:

1. Use unsafe yaml loading by default. Note that this may lead to arbitrary code execution: you must not load a config file from untrusted sources before manually inspecting the content of the file.
2. Support config versioning. When attempting to merge an old config, it will convert the old config automatically.

DEPRECATED_KEYS = '__deprecated_keys__'

IMMUTABLE = '__immutable__'

NEW_ALLOWED = '__new_allowed__'

RENAMED_KEYS = '__renamed_keys__'

__init__(*init_dict=None*, *key_list=None*, *new_allowed=False*)

Parameters

- **init_dict** (*dict*) – the possibly-nested dictionary to initailize the CfgNode.
- **key_list** (*list[str]*) – a list of names which index this CfgNode from the root. Currently only used for logging purposes.
- **new_allowed** (*bool*) – whether adding new key is allowed when merging with other configs.

classmethod _create_config_tree_from_dict(*dic*, *key_list*)

Create a configuration tree using the given dict. Any dict-like objects inside dict will be treated as a new CfgNode.

Parameters

- **dic** (*dict*) –
- **key_list** (*list[str]*) – a list of names which index this CfgNode from the root. Currently only used for logging purposes.

classmethod _decode_cfg_value(*value*)

Decodes a raw config value (e.g., from a yaml config files or command line argument) into a Python object.

If the value is a dict, it will be interpreted as a new CfgNode. If the value is a str, it will be evaluated as literals. Otherwise it is returned as-is.

_immutable(*is_immutable*)

Set immutability to is_immutable and recursively apply the setting to all nested CfgNodes.

classmethod _load_cfg_from_file(*file_obj*)

Load a config from a YAML file or a Python source file.

classmethod _load_cfg_from_yaml_str(*str_obj*)

Load a config from a YAML string encoding.

classmethod _load_cfg_py_source(*filename*)

Load a config from a Python source file.

classmethod _open_cfg(*filename*)

Defines how a config file is opened. May be overridden to support different file schemas.

clear() → None. Remove all items from D.

clone()

Recursively copy this CfgNode.

copy() → a shallow copy of D

defrost()

Make this CfgNode and all of its children mutable.

dump(**args*, ***kwargs*)

Returns
a yaml string representation of the config

Return type
str

freeze()

Make this CfgNode and all of its children immutable.

fromkeys(*value=None*, /)

Create a new dictionary with keys from iterable and values set to value.

get(*key*, *default=None*, /)

Return the value for key if key is in the dictionary, else default.

is_frozen()

Return mutability.

is_new_allowed()

items() → a set-like object providing a view on D's items

key_is_deprecated(*full_key*)

Test if a key is deprecated.

key_is_renamed(*full_key*)

Test if a key is renamed.

keys() → a set-like object providing a view on D's keys

classmethod load_cfg(*cfg_file_obj_or_str*)

Load a cfg. :param cfg_file_obj_or_str: Supports loading from:

- A file object backed by a YAML file
- A file object backed by a Python source file that exports an attribute "cfg" that is either a dict or a CfgNode
- A string that can be parsed as valid YAML

load_from_file_tmp(*cfg_filename: str*, *allow_unsafe: bool = True*) → None

classmethod load_yaml_with_base(*filename: str*, *allow_unsafe: bool = False*) → Dict[str, Any]

Just like *yaml.load(open(filename))*, but inherit attributes from its *_BASE_*.

Parameters

- **filename** (*str or file-like object*) – the file name or file of the current config. Will be used to find the base config file.
- **allow_unsafe** (*bool*) – whether to allow loading the config file with *yaml.unsafe_load*.

Returns
the loaded yaml

Return type
(dict)

merge_from_file(*cfg_filename: str*, *allow_unsafe: bool = True*) → None

Merge configs from a given yaml file.

Parameters

- **cfg_filename** – the file name of the yaml config.
- **allow_unsafe** – whether to allow loading the config file with *yaml.unsafe_load*.

merge_from_list(*cfg_list: List[str]*) → Callable[[], None]

Parameters
cfg_list (*list*) – list of configs to merge from.

merge_from_other_cfg(*cfg_other: CfgNode*) → Callable[[], None]

Parameters
cfg_other (CfgNode) – configs to merge from.

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.

If key is not found, d is returned if given, otherwise KeyError is raised

popitem() → (k, v), remove and return some (key, value) pair as a

2-tuple; but raise KeyError if D is empty.

raise_key_rename_error(*full_key*)

register_deprecated_key(*key*)

Register key (e.g. *FOO.BAR*) a deprecated option. When merging deprecated keys a warning is generated and the key is ignored.

register_renamed_key(*old_name*, *new_name*, *message=None*)

Register a key as having been renamed from *old_name* to *new_name*. When merging a renamed key, an exception is thrown alerting to user to the fact that the key has been renamed.

set_new_allowed(*is_new_allowed*)

Set this config (and recursively its subconfigs) to allow merging new keys from other configs.

setdefault(*key*, *default=None*, /)

Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

update([*E*], **F) → None. Update D from dict/iterable E and F.

If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() → an object providing a view on D's values

xmodaler.config.**get_cfg**() → *CfgNode*

Get a copy of the default config.

Returns

a X-modaler CfgNode instance.

xmodaler.config.**set_global_cfg**(*cfg:* CfgNode) → None

Let the global config point to the given cfg.

Assume that the given "cfg" has the key "KEY", after calling *set_global_cfg(cfg)*, the key can be accessed by:

```
from xmodaler.config import global_cfg
print(global_cfg.KEY)
```

By using a hacky global config, you can access these configs anywhere, without having to pass the config object or the values deep into the code. This is a hacky feature introduced for quick prototyping / research exploration.

xmodaler.config.**downgrade_config**(*cfg:* CfgNode, *to_version: int*) → *CfgNode*

Downgrade a config from its current version to an older version.

Parameters

- **cfg** (CfgNode) –
- **to_version** (*int*) –

Note: A general downgrade of arbitrary configs is not always possible due to the different functionalities in different versions. The purpose of downgrade is only to recover the defaults in old versions, allowing it to load an old partial yaml config. Therefore, the implementation only needs to fill in the default values in the old version when a general downgrade is not possible.

xmodaler.config.**upgrade_config**(*cfg:* CfgNode, *to_version: Optional[int] = None*) → *CfgNode*

Upgrade a config from its current version to a newer version.

Parameters

- **cfg** (CfgNode) –

- **to_version** (*int*) – defaults to the latest version.

xmodaler.config.**configurable**(*init_func=None*, *, *from_config=None*)

Decorate a function or a class's __init__ method so that it can be called with a *CfgNode* object using a from_config() function that translates *CfgNode* to arguments.

Examples:

```
# Usage 1: Decorator on __init__:
class A:
    @configurable
    def __init__(self, a, b=2, c=3):
        pass

    @classmethod
    def from_config(cls, cfg):   # 'cfg' must be the first argument
        # Returns kwargs to be passed to __init__
        return {"a": cfg.A, "b": cfg.B}

a1 = A(a=1, b=2)  # regular construction
a2 = A(cfg)       # construct with a cfg
a3 = A(cfg, b=3, c=4)  # construct with extra overwrite

# Usage 2: Decorator on any function. Needs an extra from_config argument:
@configurable(from_config=lambda cfg: {"a: cfg.A, "b": cfg.B})
def a_func(a, b=2, c=3):
    pass

a1 = a_func(a=1, b=2)  # regular call
a2 = a_func(cfg)       # call with a cfg
a3 = a_func(cfg, b=3, c=4)  # call with extra overwrite
```

Parameters

- **init_func** (*callable*) – a class's __init__ method in usage 1. The class must have a from_config classmethod which takes *cfg* as the first argument.
- **from_config** (*callable*) – the from_config function in usage 2. It must take *cfg* as its first argument.

xmodaler.config.**global_cfg**

The same as *fvcore.common.config.CfgNode*, but different in:

1. Use unsafe yaml loading by default. Note that this may lead to arbitrary code execution: you must not load a config file from untrusted sources before manually inspecting the content of the file.
2. Support config versioning. When attempting to merge an old config, it will convert the old config automatically.

xmodaler.config.**kfg**

The same as *fvcore.common.config.CfgNode*, but different in:

1. Use unsafe yaml loading by default. Note that this may lead to arbitrary code execution: you must not load a config file from untrusted sources before manually inspecting the content of the file.
2. Support config versioning. When attempting to merge an old config, it will convert the old config automatically.

3.3 xmodaler.datasets

class xmodaler.datasets.**DatasetFromList**(*lst: list*, *copy: bool = True*, *serialize: bool = True*)

Bases: Dataset

Wrap a list to a torch Dataset. It produces elements of the list as data.

__getitem__(*idx*)

__init__(*lst: list*, *copy: bool = True*, *serialize: bool = True*)

Parameters

- **lst** (*list*) – a list which contains elements to produce.
- **copy** (*bool*) – whether to deepcopy the element when producing it, so that the result can be modified in place without affecting the source in the list.
- **serialize** (*bool*) – whether to hold memory using serialized objects, when enabled, data loader workers can use shared RAM from master process instead of making a copy.

__len__()

class xmodaler.datasets.**MapDataset**(*dataset*, *map_func*)

Bases: Dataset

Map a function over the elements in a dataset.

Parameters

- **dataset** – a dataset where map function is applied.
- **map_func** – a callable which maps the element in dataset. map_func is responsible for error handling, when error happens, it needs to return None so the MapDataset will randomly use other elements from the dataset.

__getitem__(*idx*)

__init__(*dataset*, *map_func*)

__len__()

class xmodaler.datasets.**MSCoCoDataset**(*stage: str*, *anno_file: str*, *seq_per_img: int*, *max_feat_num: int*, *max_seq_len: int*, *feats_folder: str*, *relation_file: str*, *gv_feat_file: str*, *attribute_file: str*)

Bases: object

__call__(*dataset_dict*)

Call self as a function.

__init__(*stage: str*, *anno_file: str*, *seq_per_img: int*, *max_feat_num: int*, *max_seq_len: int*, *feats_folder: str*, *relation_file: str*, *gv_feat_file: str*, *attribute_file: str*)

classmethod from_config(*cfg*, *stage: str = 'train'*)

load_data(*cfg*)

class xmodaler.datasets.**MSCoCoSampleByTxtDataset**(*stage: str*, *anno_file: str*, *seq_per_img: int*, *max_feat_num: int*, *max_seq_len: int*, *feats_folder: str*, *relation_file: str*, *gv_feat_file: str*, *attribute_file: str*)

Bases: *MSCoCoDataset*

__call__(*dataset_dict*)

Call self as a function.

__init__(*stage: str*, *anno_file: str*, *seq_per_img: int*, *max_feat_num: int*, *max_seq_len: int*, *feats_folder: str*, *relation_file: str*, *gv_feat_file: str*, *attribute_file: str*)

classmethod from_config(*cfg*, *stage: str = 'train'*)

load_data(*cfg*)

class xmodaler.datasets.**MSCoCoBertDataset**(*stage: str*, *anno_file: str*, *seq_per_img: int*, *max_seq_length: int*, *max_feat_num: int*, *feats_folder: str*, *images_ids_file: str*, *tokenizer*)

Bases: `object`

__call__(*dataset_dict*)

Call self as a function.

__init__(*stage: str*, *anno_file: str*, *seq_per_img: int*, *max_seq_length: int*, *max_feat_num: int*, *feats_folder: str*, *images_ids_file: str*, *tokenizer*)

classmethod from_config(*cfg*, *stage: str = 'train'*)

load_data(*cfg*)

class xmodaler.datasets.**ConceptualCaptionsDataset**(*stage: str*, *anno_file: str*, *max_seq_length: int*, *max_feat_num: int*, *feats_folder: str*, *images_ids_file: str*, *tokenizer*)

Bases: `object`

__call__(*dataset_dict*)

Call self as a function.

__init__(*stage: str*, *anno_file: str*, *max_seq_length: int*, *max_feat_num: int*, *feats_folder: str*, *images_ids_file: str*, *tokenizer*)

classmethod from_config(*cfg*, *stage: str = 'train'*)

load_data(*cfg*)

class xmodaler.datasets.**ConceptualCaptionsDatasetForSingleStream**(*stage: str*, *anno_file: str*, *max_seq_length: int*, *max_feat_num: int*, *feats_folder: str*, *images_ids_file: str*, *tokenizer*, *itm_neg_prob: float*)

Bases: *ConceptualCaptionsDataset*

__call__(*dataset_dict*)

Call self as a function.

__init__(*stage: str*, *anno_file: str*, *max_seq_length: int*, *max_feat_num: int*, *feats_folder: str*, *images_ids_file: str*, *tokenizer*, *itm_neg_prob: float*)

classmethod from_config(*cfg*, *stage: str = 'train'*)

load_data(*cfg*)

class xmodaler.datasets.**VQADataset**(*stage: str*, *anno_folder: str*, *ans2label_path: str*, *label2ans_path: str*, *feats_folder: str*, *max_feat_num: int*, *max_seq_len: int*, *use_global_v: bool*, *tokenizer*)

Bases: `object`

__call__(*dataset_dict*)

Call self as a function.

__init__(*stage: str*, *anno_folder: str*, *ans2label_path: str*, *label2ans_path: str*, *feats_folder: str*, *max_feat_num: int*, *max_seq_len: int*, *use_global_v: bool*, *tokenizer*)

classmethod from_config(*cfg*, *stage: str = 'train'*)

load_data(*cfg*)

class xmodaler.datasets.**VCRDataset**(*stage: str*, *task_name: str*, *anno_folder: str*, *feats_folder: str*, *max_feat_num: int*, *max_seq_len: int*, *seq_per_img: int*, *use_global_v: bool*, *tokenizer*)

Bases: `object`

__call__(*dataset_dict*)

Call self as a function.

__init__(*stage: str*, *task_name: str*, *anno_folder: str*, *feats_folder: str*, *max_feat_num: int*, *max_seq_len: int*, *seq_per_img: int*, *use_global_v: bool*, *tokenizer*)

classmethod from_config(*cfg*, *stage: str = 'train;VCR_Q-A'*)

load_data(*cfg*)

class xmodaler.datasets.**Flickr30kDataset**(*stage: str*, *anno_folder: str*, *anno_file: str*, *feats_folder: str*, *max_feat_num: int*, *max_seq_len: int*, *use_global_v: bool*, *tokenizer*)

Bases: `object`

__call__(*dataset_dict*)

Call self as a function.

__init__(*stage: str*, *anno_folder: str*, *anno_file: str*, *feats_folder: str*, *max_feat_num: int*, *max_seq_len: int*, *use_global_v: bool*, *tokenizer*)

classmethod from_config(*cfg*, *stage: str = 'train'*)

load_data(*cfg*)

class xmodaler.datasets.**Flickr30kDatasetForSingleStream**(*stage: str*, *anno_folder: str*, *anno_file: str*, *feats_folder: str*, *max_feat_num: int*, *max_seq_len: int*, *use_global_v: bool*, *negative_size: int*, *tokenizer*, *cfg*)

Bases: `Flickr30kDataset`

__call__(*dataset_dict*)

Call self as a function.

__init__(*stage: str*, *anno_folder: str*, *anno_file: str*, *feats_folder: str*, *max_feat_num: int*, *max_seq_len: int*, *use_global_v: bool*, *negative_size: int*, *tokenizer*, *cfg*)

classmethod from_config(*cfg*, *stage: str = 'train'*)

load_data(*cfg*)

class xmodaler.datasets.**Flickr30kDatasetForSingleStreamVal**(*stage: str*, *anno_folder: str*, *anno_file: str*, *feats_folder: str*, *max_feat_num: int*, *max_seq_len: int*, *use_global_v: bool*, *inf_batch_size: int*, *tokenizer*, *cfg*)

Bases: *Flickr30kDataset*

__call__(*dataset_dict*)

Call self as a function.

__init__(*stage: str*, *anno_folder: str*, *anno_file: str*, *feats_folder: str*, *max_feat_num: int*, *max_seq_len: int*, *use_global_v: bool*, *inf_batch_size: int*, *tokenizer*, *cfg*)

classmethod from_config(*cfg*, *stage: str = 'train'*)

load_data(*cfg*)

class xmodaler.datasets.**MSVDDataset**(*stage: str*, *anno_file: str*, *seq_per_img: int*, *max_feat_num: int*, *max_seq_len: int*, *feats_folder: str*)

Bases: `object`

__call__(*dataset_dict*)

Call self as a function.

__init__(*stage: str*, *anno_file: str*, *seq_per_img: int*, *max_feat_num: int*, *max_seq_len: int*, *feats_folder: str*)

classmethod from_config(*cfg*, *stage: str = 'train'*)

load_data(*cfg*)

class xmodaler.datasets.**MSRVTTDataset**(*stage: str*, *anno_file: str*, *seq_per_img: int*, *max_feat_num: int*, *max_seq_len: int*, *feats_folder: str*)

Bases: `object`

__call__(*dataset_dict*)

Call self as a function.

__init__(*stage: str*, *anno_file: str*, *seq_per_img: int*, *max_feat_num: int*, *max_seq_len: int*, *feats_folder: str*)

classmethod from_config(*cfg*, *stage: str = 'train'*)

load_data(*cfg*)

xmodaler.datasets.**build_xmodaler_train_loader**(*datalist*, ***, *dataset_mapper*, *batch_size*, *num_workers*)

`xmodaler.datasets.build_xmodaler_valtest_loader(datalist, *, dataset_mapper, batch_size, num_workers, multi_gpu_eval)`

`xmodaler.datasets.build_dataset_mapper(cfg, name, stage)`

3.4 xmodaler.engine

`xmodaler.engine.launch(main_func, num_gpus_per_machine, num_machines=1, machine_rank=0, dist_url=None, args=(), timeout=datetime.timedelta(seconds=1800))`

Launch multi-gpu or distributed training. This function must be called on all machines involved in the training. It will spawn child processes (defined by `num_gpus_per_machine`) on each machine.

Parameters

- **main_func** – a function that will be called by *main_func(*args)*
- **num_gpus_per_machine** (*int*) – number of GPUs per machine
- **num_machines** (*int*) – the total number of machines
- **machine_rank** (*int*) – the rank of this machine
- **dist_url** (*str*) – url to connect to for distributed jobs, including protocol e.g. "tcp://127.0.0.1:8686". Can be set to "auto" to automatically select a free port on localhost
- **timeout** (*timedelta*) – timeout of the distributed workers
- **args** (*tuple*) – arguments passed to main_func

`xmodaler.engine.build_engine(cfg)`

`xmodaler.engine.default_argument_parser(epilog=None)`

Create a parser with some common arguments used by X-modaler users.

Parameters
epilog (*str*) – epilog passed to ArgumentParser describing the usage.

Return type
argparse.ArgumentParser

`xmodaler.engine.default_setup(cfg, args)`

Perform some basic common setups at the beginning of a job, including:

1. Set up the X-modaler logger
2. Log basic information about environment, cmdline arguments, and config
3. Backup the config to the output directory

Parameters

- **cfg** (`CfgNode`) – the full config to be used
- **args** (*argparse.NameSpace*) – the command line arguments to be logged

`xmodaler.engine.default_writers(output_dir: str, max_iter: Optional[int] = None)`

Build a list of `EventWriter` to be used. It now consists of a `CommonMetricPrinter`, `TensorboardXWriter` and `JSONWriter`.

Parameters

- **output_dir** – directory to store JSON metrics and tensorboard events
- **max_iter** – the total number of iterations

Returns
a list of `EventWriter` objects.

Return type
list[EventWriter]

class `xmodaler.engine.`**`DefaultTrainer`**(*cfg*)

Bases: *TrainerBase*

A trainer with default training logic. It does the following:

1. Create a *DefaultTrainer* using model, optimizer, dataloader defined by the given config. Create a LR scheduler defined by the config.
2. Load the last checkpoint or *cfg.MODEL.WEIGHTS*, if exists, when *resume_or_load* is called.
3. Register a few common hooks defined by the config.

It is created to simplify the **standard model training workflow** and reduce code boilerplate for users who only need the standard training workflow, with standard features. It means this class makes *many assumptions* about your training logic that may easily become invalid in a new research. In fact, any assumptions beyond those made in the *DefaultTrainer* are too much for research.

The code of this class has been annotated about restrictive assumptions it makes. When they do not work for you, you're encouraged to:

1. Overwrite methods of this class, OR:
2. Use *DefaultTrainer*, which only does minimal SGD training and nothing else. You can then add your own hooks if needed. OR:
3. Write your own training loop similar to *train_net.py*.

See the *Training* tutorials for more details.

Note that the behavior of this class, like other functions/classes in this file, is not stable, since it is meant to represent the "common default behavior". It is only guaranteed to work well with the standard models and training workflow in X-modaler. To obtain more stable behavior, write your own training logic with other public APIs.

Examples:

```
trainer = DefaultTrainer(cfg)
trainer.resume_or_load()  # load last checkpoint or MODEL.WEIGHTS
trainer.train()
```

scheduler

checkpointer

Type
XmodalerCheckpointer

cfg

Type
CfgNode

__init__(*cfg*)

Parameters
cfg (CfgNode) –

_write_metrics(*loss_dict: Dict[str, Tensor]*, *data_time: float*, *prefix: str = ''*)

Parameters

- **loss_dict** (*dict*) – dict of scalar losses
- **data_time** (*float*) – time taken by the dataloader iteration

after_step()

after_train()

static auto_scale_workers(*cfg*, *num_workers: int*)

before_step()

before_train()

build_hooks()

classmethod build_losses(*cfg*)

classmethod build_lr_scheduler(*cfg*, *optimizer*, *iters_per_epoch*)

classmethod build_model(*cfg*)

classmethod build_optimizer(*cfg*, *model*)

classmethod build_test_loader(*cfg*)

classmethod build_train_loader(*cfg*)

classmethod build_val_loader(*cfg*)

build_writers()

load_state_dict(*state_dict*)

register_hooks(*hooks: List[Optional[*HookBase*]]*) → None

Register hooks to the trainer. The hooks are executed in the order they are registered.

Parameters
hooks (*list[Optional[*HookBase*]]*) – list of hooks

resume_or_load(*resume=True*)

run_step()

Implement the standard training logic described above.

state_dict()

classmethod test(*cfg*, *model*, *test_data_loader*, *evaluator*, *epoch*)

train()

Parameters

- **start_iter** (*int*) – See docs above
- **max_iter** (*int*) – See docs above

class xmodaler.engine.**HookBase**

Bases: object

Base class for hooks that can be registered with *TrainerBase*.

Each hook can implement 4 methods. The way they are called is demonstrated in the following snippet:

```
hook.before_train()
for iter in range(start_iter, max_iter):
    hook.before_step()
    trainer.run_step()
    hook.after_step()
iter += 1
hook.after_train()
```

Notes

1. In the hook method, users can access `self.trainer` to access more properties about the context (e.g., model, current iteration, or config if using *DefaultTrainer*).
2. A hook that does something in *before_step()* can often be implemented equivalently in *after_step()*. If the hook takes non-trivial time, it is strongly recommended to implement the hook in *after_step()* instead of *before_step()*. The convention is that *before_step()* should only take negligible time.

   Following this convention will allow hooks that do care about the difference between *before_step()* and *after_step()* (e.g., timer) to function properly.

after_step()

Called after each iteration.

after_train()

Called after the last iteration.

before_step()

Called before each iteration.

before_train()

Called before the first iteration.

state_dict()

Hooks are stateless by default, but can be made checkpointable by implementing *state_dict* and *load_state_dict*.

trainer: *TrainerBase* = None

A weak reference to the trainer object. Set by the trainer when the hook is registered.

class xmodaler.engine.**TrainerBase**

Bases: object

Base class for iterative trainer with hooks.

The only assumption we made here is: the training runs in a loop. A subclass can implement what the loop is. We made no assumptions about the existence of dataloader, optimizer, model, etc.

iter

the current iteration.

Type
int

start_iter

The iteration to start with. By convention the minimum possible value is 0.

Type
int

max_iter

The iteration to end training.

Type
int

storage

An EventStorage that's opened during the course of training.

Type
EventStorage

__init__() → None

after_step()

after_train()

before_step()

before_train()

load_state_dict(*state_dict*)

register_hooks(*hooks: List[Optional[*HookBase*]]*) → None

Register hooks to the trainer. The hooks are executed in the order they are registered.

Parameters
hooks (*list[Optional[*HookBase*]]*) – list of hooks

run_step()

state_dict()

train(*start_iter: int*, *max_iter: int*)

Parameters

- **start_iter** (*int*) – See docs above
- **max_iter** (*int*) – See docs above

class xmodaler.engine.**CallbackHook**(*, *before_train=None*, *after_train=None*, *before_step=None*, *after_step=None*)

Bases: *HookBase*

Create a hook using callback functions provided by the user.

__init__(*, *before_train=None*, *after_train=None*, *before_step=None*, *after_step=None*)

Each argument is a function that takes one argument: the trainer.

after_step()

Called after each iteration.

after_train()

Called after the last iteration.

before_step()

Called before each iteration.

before_train()

Called before the first iteration.

state_dict()

Hooks are stateless by default, but can be made checkpointable by implementing *state_dict* and *load_state_dict*.

trainer: *TrainerBase* = None

A weak reference to the trainer object. Set by the trainer when the hook is registered.

class xmodaler.engine.**IterationTimer**(*warmup_iter=3*)

Bases: *HookBase*

Track the time spent for each iteration (each run_step call in the trainer). Print a summary in the end of training.

This hook uses the time between the call to its *before_step()* and *after_step()* methods. Under the convention that *before_step()* of all hooks should only take negligible amount of time, the *IterationTimer* hook should be placed at the beginning of the list of hooks to obtain accurate timing.

__init__(*warmup_iter=3*)

Parameters

warmup_iter (*int*) – the number of iterations at the beginning to exclude from timing.

after_step()

Called after each iteration.

after_train()

Called after the last iteration.

before_step()

Called before each iteration.

before_train()

Called before the first iteration.

state_dict()

Hooks are stateless by default, but can be made checkpointable by implementing *state_dict* and *load_state_dict*.

trainer: *TrainerBase* = **None**

A weak reference to the trainer object. Set by the trainer when the hook is registered.

class xmodaler.engine.**PeriodicWriter**(*writers*, *period=20*)

Bases: *HookBase*

Write events to EventStorage (by calling `writer.write()`) periodically.

It is executed every `period` iterations and after the last iteration. Note that `period` does not affect how data is smoothed by each writer.

__init__(*writers*, *period=20*)

Parameters

- **writers** (*list[EventWriter]*) – a list of EventWriter objects
- **period** (*int*) –

after_step()

Called after each iteration.

after_train()

Called after the last iteration.

before_step()

Called before each iteration.

before_train()

Called before the first iteration.

state_dict()

Hooks are stateless by default, but can be made checkpointable by implementing *state_dict* and *load_state_dict*.

trainer: *TrainerBase* = **None**

A weak reference to the trainer object. Set by the trainer when the hook is registered.

class xmodaler.engine.**PeriodicCheckpointer**(*checkpointer: Checkpointer*, *period: int*, *max_iter: Optional[int] = None*, *max_to_keep: Optional[int] = None*, *file_prefix: str = 'model'*)

Bases: *PeriodicEpochCheckpointer*, *HookBase*

Same as `detectron2.checkpoint.PeriodicCheckpointer`, but as a hook.

Note that when used as a hook, it is unable to save additional data other than what's defined by the given *checkpointer*.

It is executed every `period` iterations and after the last iteration.

__init__(*checkpointer: Checkpointer*, *period: int*, *max_iter: Optional[int] = None*, *max_to_keep: Optional[int] = None*, *file_prefix: str = 'model'*) → None

Parameters

- **checkpointer** – the checkpointer object used to save checkpoints.
- **period** (*int*) – the period to save checkpoint.
- **max_iter** (*int*) – maximum number of iterations. When it is reached, a checkpoint named "{file_prefix}_final" will be saved.

- **max_to_keep** (*int*) – maximum number of most current checkpoints to keep, previous checkpoints will be deleted
- **file_prefix** (*str*) – the prefix of checkpoint's filename

after_step()

Called after each iteration.

after_train()

Called after the last iteration.

before_step()

Called before each iteration.

before_train()

Called before the first iteration.

save(*name: str*, ***kwargs: Any*) → None

Same argument as `Checkpointer.save()`. Use this method to manually save checkpoints outside the schedule.

Parameters

- **name** (*str*) – file name.
- **kwargs** (*Any*) – extra data to save, same as in `Checkpointer.save()`.

state_dict()

Hooks are stateless by default, but can be made checkpointable by implementing *state_dict* and *load_state_dict*.

step(*iteration: int*, *epoch: int*, ***kwargs: Any*) → None

Perform the appropriate action at the given iteration.

Parameters

- **iteration** (*int*) – the current iteration, ranged in [0, max_iter-1].
- **kwargs** (*Any*) – extra data to save, same as in `Checkpointer.save()`.

trainer: *TrainerBase* = None

A weak reference to the trainer object. Set by the trainer when the hook is registered.

class xmodaler.engine.**LRScheduler**(*optimizer=None*, *scheduler=None*)

Bases: *HookBase*

A hook which executes a torch builtin LR scheduler and summarizes the LR. It is executed after every iteration.

__init__(*optimizer=None*, *scheduler=None*)

Parameters

- **optimizer** (*torch.optim.Optimizer*) –
- **scheduler** (*torch.optim.LRScheduler*) –

If any argument is not given, will try to obtain it from the trainer.

after_step()

Called after each iteration.

after_train()

Called after the last iteration.

before_step()

Called before each iteration.

before_train()

Called before the first iteration.

state_dict()

Hooks are stateless by default, but can be made checkpointable by implementing *state_dict* and *load_state_dict*.

trainer: *TrainerBase* = None

A weak reference to the trainer object. Set by the trainer when the hook is registered.

class xmodaler.engine.**AutogradProfiler**(*enable_predicate*, *output_dir*, *, *use_cuda=True*)

Bases: *HookBase*

A hook which runs *torch.autograd.profiler.profile*.

Examples:

```
hooks.AutogradProfiler(
     lambda trainer: trainer.iter > 10 and trainer.iter < 20, self.cfg.OUTPUT_DIR
)
```

The above example will run the profiler for iteration 10~20 and dump results to `OUTPUT_DIR`. We did not profile the first few iterations because they are typically slower than the rest. The result files can be loaded in the `chrome://tracing` page in chrome browser.

Note: When used together with NCCL on older version of GPUs, autograd profiler may cause deadlock because it unnecessarily allocates memory on every device it sees. The memory management calls, if interleaved with NCCL calls, lead to deadlock on GPUs that do not support `cudaLaunchCooperativeKernelMultiDevice`.

__init__(*enable_predicate*, *output_dir*, *, *use_cuda=True*)

Parameters

- **enable_predicate** (*callable[trainer -> bool]*) – a function which takes a trainer, and returns whether to enable the profiler. It will be called once every step, and can be used to select which steps to profile.
- **output_dir** (*str*) – the output directory to dump tracing files.
- **use_cuda** (*bool*) – same as in *torch.autograd.profiler.profile*.

after_step()

Called after each iteration.

after_train()

Called after the last iteration.

before_step()

Called before each iteration.

before_train()

Called before the first iteration.

state_dict()

Hooks are stateless by default, but can be made checkpointable by implementing *state_dict* and *load_state_dict*.

trainer: TrainerBase = None

A weak reference to the trainer object. Set by the trainer when the hook is registered.

class xmodaler.engine.**EvalHook**(*eval_period*, *eval_start*, *eval_function*, *iters_per_epoch*, *stage*, *multi_gpu_eval*)

Bases: *HookBase*

Run an evaluation function periodically, and at the end of training.

It is executed every `eval_period` iterations and after the last iteration.

__init__(*eval_period*, *eval_start*, *eval_function*, *iters_per_epoch*, *stage*, *multi_gpu_eval*)

Parameters

- **eval_period** (*int*) – the period to run *eval_function*. Set to 0 to not evaluate periodically (but still after the last iteration).
- **eval_function** (*callable*) – a function which takes no arguments, and returns a nested dict of evaluation metrics.

Note: This hook must be enabled in all or none workers. If you would like only certain workers to perform evaluation, give other workers a no-op function (*eval_function=lambda: None*).

_do_eval(*epoch*)

after_step()

Called after each iteration.

after_train()

Called after the last iteration.

before_step()

Called before each iteration.

before_train()

Called before the first iteration.

state_dict()

Hooks are stateless by default, but can be made checkpointable by implementing *state_dict* and *load_state_dict*.

trainer: TrainerBase = None

A weak reference to the trainer object. Set by the trainer when the hook is registered.

class xmodaler.engine.**PreciseBN**(*period*, *model*, *data_loader*, *num_iter*)

Bases: *HookBase*

The standard implementation of BatchNorm uses EMA in inference, which is sometimes suboptimal. This class computes the true average of statistics rather than the moving average, and put true averages to every BN layer in the given model.

It is executed every `period` iterations and after the last iteration.

__init__(*period*, *model*, *data_loader*, *num_iter*)

Parameters

- **period** (*int*) – the period this hook is run, or 0 to not run during training. The hook will always run in the end of training.
- **model** (*nn.Module*) – a module whose all BN layers in training mode will be updated by precise BN. Note that user is responsible for ensuring the BN layers to be updated are in training mode when this hook is triggered.
- **data_loader** (*iterable*) – it will produce data to be run by *model(data)*.
- **num_iter** (*int*) – number of iterations used to compute the precise statistics.

after_step()

Called after each iteration.

after_train()

Called after the last iteration.

before_step()

Called before each iteration.

before_train()

Called before the first iteration.

state_dict()

Hooks are stateless by default, but can be made checkpointable by implementing *state_dict* and *load_state_dict*.

trainer: *TrainerBase* = None

A weak reference to the trainer object. Set by the trainer when the hook is registered.

update_stats()

Update the model with precise statistics. Users can manually call this method.

class xmodaler.engine.**ModelWeightsManipulating**

Bases: *HookBase*

Init or bind weights after loading a model

__init__()

after_step()

Called after each iteration.

after_train()

Called after the last iteration.

before_step()

Called before each iteration.

before_train()

Called before the first iteration.

state_dict()

Hooks are stateless by default, but can be made checkpointable by implementing *state_dict* and *load_state_dict*.

trainer: *TrainerBase* = None

A weak reference to the trainer object. Set by the trainer when the hook is registered.

class xmodaler.engine.**RetrievalTrainer**(*cfg*)

Bases: *DefaultTrainer*

__init__(*cfg*)

Parameters

cfg (CfgNode) –

_write_metrics(*loss_dict: Dict[str, Tensor]*, *data_time: float*, *prefix: str = ''*)

Parameters

- **loss_dict** (*dict*) – dict of scalar losses
- **data_time** (*float*) – time taken by the dataloader iteration

after_step()

after_train()

static auto_scale_workers(*cfg*, *num_workers: int*)

before_step()

before_train()

build_hooks()

classmethod build_losses(*cfg*)

classmethod build_lr_scheduler(*cfg*, *optimizer*, *iters_per_epoch*)

classmethod build_model(*cfg*)

classmethod build_optimizer(*cfg*, *model*)

classmethod build_test_loader(*cfg*)

classmethod build_train_loader(*cfg*)

classmethod build_val_loader(*cfg*)

build_writers()

load_state_dict(*state_dict*)

register_hooks(*hooks: List[Optional[*HookBase*]]*) → None

Register hooks to the trainer. The hooks are executed in the order they are registered.

Parameters

hooks (*list[Optional[*HookBase*]]*) – list of hooks

resume_or_load(*resume=True*)

run_step()

Implement the standard training logic described above.

state_dict()

classmethod test(*cfg*, *model*, *test_data_loader*, *evaluator*, *epoch*)

train()

Parameters

- **start_iter** (*int*) – See docs above
- **max_iter** (*int*) – See docs above

class xmodaler.engine.**RLTrainer**(*cfg*)

Bases: *DefaultTrainer*

__init__(*cfg*)

Parameters

cfg (CfgNode) –

_write_metrics(*loss_dict: Dict[str, Tensor]*, *data_time: float*, *prefix: str = ''*)

Parameters

- **loss_dict** (*dict*) – dict of scalar losses
- **data_time** (*float*) – time taken by the dataloader iteration

after_step()

after_train()

static auto_scale_workers(*cfg*, *num_workers: int*)

before_step()

before_train()

build_hooks()

classmethod build_losses(*cfg*)

classmethod build_lr_scheduler(*cfg*, *optimizer*, *iters_per_epoch*)

classmethod build_model(*cfg*)

classmethod build_optimizer(*cfg*, *model*)

classmethod build_scorer(*cfg*)

classmethod build_test_loader(*cfg*)

classmethod build_train_loader(*cfg*)

classmethod build_val_loader(*cfg*)

build_writers()

load_state_dict(*state_dict*)

register_hooks(*hooks: List[Optional[*HookBase*]]*) → None

Register hooks to the trainer. The hooks are executed in the order they are registered.

Parameters

hooks (*list[Optional[*HookBase*]]*) – list of hooks

resume_or_load(*resume=True*)

run_step()

Implement the standard training logic described above.

state_dict()

classmethod test(*cfg*, *model*, *test_data_loader*, *evaluator*, *epoch*)

train()

Parameters

- **start_iter** (*int*) – See docs above
- **max_iter** (*int*) – See docs above

class xmodaler.engine.**RLBeamTrainer**(*cfg*)

Bases: *DefaultTrainer*

__init__(*cfg*)

Parameters

cfg (CfgNode) –

_write_metrics(*loss_dict: Dict[str, Tensor]*, *data_time: float*, *prefix: str = ''*)

Parameters

- **loss_dict** (*dict*) – dict of scalar losses
- **data_time** (*float*) – time taken by the dataloader iteration

after_step()

after_train()

static auto_scale_workers(*cfg*, *num_workers: int*)

before_step()

before_train()

build_hooks()

classmethod build_losses(*cfg*)

classmethod build_lr_scheduler(*cfg*, *optimizer*, *iters_per_epoch*)

classmethod build_model(*cfg*)

classmethod build_optimizer(*cfg*, *model*)

classmethod build_scorer(*cfg*)

classmethod build_test_loader(*cfg*)

classmethod build_train_loader(*cfg*)

classmethod build_val_loader(*cfg*)

build_writers()

load_state_dict(*state_dict*)

register_hooks(*hooks: List[Optional[HookBase]]*) → None

Register hooks to the trainer. The hooks are executed in the order they are registered.

Parameters

hooks (*list[Optional[HookBase]]*) – list of hooks

resume_or_load(*resume=True*)

run_step()

Implement the standard training logic described above.

state_dict()

classmethod test(*cfg*, *model*, *test_data_loader*, *evaluator*, *epoch*)

train()

Parameters

- **start_iter** (*int*) – See docs above
- **max_iter** (*int*) – See docs above

class xmodaler.engine.**SingleStreamRetrievalTrainer**(*cfg*)

Bases: *DefaultTrainer*

__init__(*cfg*)

Parameters

cfg (CfgNode) –

_write_metrics(*loss_dict: Dict[str, Tensor]*, *data_time: float*, *prefix: str = ''*)

Parameters

- **loss_dict** (*dict*) – dict of scalar losses
- **data_time** (*float*) – time taken by the dataloader iteration

after_step()

after_train()

static auto_scale_workers(*cfg*, *num_workers: int*)

before_step()

before_train()

build_hooks()

classmethod build_losses(*cfg*)

classmethod build_lr_scheduler(*cfg*, *optimizer*, *iters_per_epoch*)

classmethod build_model(*cfg*)

classmethod build_optimizer(*cfg*, *model*)

classmethod build_test_loader(*cfg*)

classmethod build_train_loader(*cfg*)

classmethod build_val_loader(*cfg*)

build_writers()

load_state_dict(*state_dict*)

register_hooks(*hooks: List[Optional[*HookBase*]]*) → None

Register hooks to the trainer. The hooks are executed in the order they are registered.

Parameters

hooks (*list[Optional[*HookBase*]]*) – list of hooks

resume_or_load(*resume=True*)

run_step()

Implement the standard training logic described above.

state_dict()

classmethod test(*cfg*, *model*, *test_data_loader*, *evaluator*, *epoch*)

train()

Parameters

- **start_iter** (*int*) – See docs above
- **max_iter** (*int*) – See docs above

class xmodaler.engine.**SingleStreamRetrievalTrainerHardNegatives**(*cfg*)

Bases: *SingleStreamRetrievalTrainer*

__init__(*cfg*)

Parameters

cfg (CfgNode) –

_write_metrics(*loss_dict: Dict[str, Tensor]*, *data_time: float*, *prefix: str = ''*)

Parameters

- **loss_dict** (*dict*) – dict of scalar losses
- **data_time** (*float*) – time taken by the dataloader iteration

after_step()

after_train()

static auto_scale_workers(*cfg*, *num_workers: int*)

before_step()

before_train()

build_hooks()

classmethod build_losses(*cfg*)

classmethod build_lr_scheduler(*cfg*, *optimizer*, *iters_per_epoch*)

classmethod build_model(*cfg*)

classmethod build_optimizer(*cfg*, *model*)

classmethod build_test_loader(*cfg*)

classmethod build_train_loader(*cfg*)

classmethod build_val_loader(*cfg*)

build_writers()

clip_inputs(*data*)

hard_negative_mining(*data*)

load_state_dict(*state_dict*)

register_hooks(*hooks: List[Optional[*HookBase*]]*) → None

Register hooks to the trainer. The hooks are executed in the order they are registered.

Parameters

hooks (*list[Optional[*HookBase*]]*) – list of hooks

resume_or_load(*resume=True*)

run_step()

Implement the standard training logic described above.

state_dict()

classmethod test(*cfg*, *model*, *test_data_loader*, *evaluator*, *epoch*)

train()

Parameters

- **start_iter** (*int*) – See docs above
- **max_iter** (*int*) – See docs above

class xmodaler.engine.**TDENPretrainer**(*cfg*)

Bases: *DefaultTrainer*

__init__(*cfg*)

Parameters

cfg (CfgNode) –

_write_metrics(*loss_dict: Dict[str, Tensor]*, *data_time: float*, *prefix: str = ''*)

Parameters

- **loss_dict** (*dict*) – dict of scalar losses
- **data_time** (*float*) – time taken by the dataloader iteration

after_step()

after_train()

static auto_scale_workers(*cfg*, *num_workers: int*)

before_step()

before_train()

build_hooks()

classmethod build_losses(*cfg*)

classmethod build_lr_scheduler(*cfg*, *optimizer*, *iters_per_epoch*)

classmethod build_model(*cfg*)

classmethod build_optimizer(*cfg*, *model*)

classmethod build_test_loader(*cfg*)

classmethod build_train_loader(*cfg*)

classmethod build_val_loader(*cfg*)

build_writers()

load_state_dict(*state_dict*)

register_hooks(*hooks: List[Optional[*HookBase*]]*) → None

Register hooks to the trainer. The hooks are executed in the order they are registered.

Parameters

hooks (*list[Optional[*HookBase*]]*) – list of hooks

resume_or_load(*resume=True*)

run_step()

Implement the standard training logic described above.

state_dict()

classmethod test(*cfg*, *model*, *test_data_loader*, *evaluator*, *epoch*)

train()

Parameters

- **start_iter** (*int*) – See docs above
- **max_iter** (*int*) – See docs above

class xmodaler.engine.**VCRTrainer**(*cfg*)

Bases: *DefaultTrainer*

__init__(*cfg*)

Parameters

cfg (CfgNode) –

_write_metrics(*loss_dict: Dict[str, Tensor]*, *data_time: float*, *prefix: str = ''*)

Parameters

- **loss_dict** (*dict*) – dict of scalar losses
- **data_time** (*float*) – time taken by the dataloader iteration

after_step()

after_train()

static auto_scale_workers(*cfg*, *num_workers: int*)

before_step()

before_train()

build_hooks()

classmethod build_losses(*cfg*)

classmethod build_lr_scheduler(*cfg*, *optimizer*, *iters_per_epoch*)

classmethod build_model(*cfg*)

classmethod build_optimizer(*cfg*, *model*)

classmethod build_test_loader(*cfg*)

classmethod build_train_loader(*cfg*)

classmethod build_val_loader(*cfg*)

build_writers()

load_state_dict(*state_dict*)

register_hooks(*hooks: List[Optional[*HookBase*]]*) → None

Register hooks to the trainer. The hooks are executed in the order they are registered.

Parameters

hooks (*list[Optional[*HookBase*]]*) – list of hooks

resume_or_load(*resume=True*)

run_step()

Implement the standard training logic described above.

state_dict()

classmethod test(*cfg*, *model*, *test_data_loader*, *evaluator*, *epoch*)

train()

Parameters

- **start_iter** (*int*) – See docs above
- **max_iter** (*int*) – See docs above

3.5 xmodaler.evaluation

xmodaler.evaluation.**build_evaluation**(*cfg*, *annfile*, *output_dir*)

class xmodaler.evaluation.**COCOEvaler**(*cfg*, *annfile*, *output_dir*)

Bases: object

__init__(*cfg*, *annfile*, *output_dir*)

eval(*results*, *epoch*)

class xmodaler.evaluation.**VQAEvaler**(*cfg*, *annfile*, *output_dir*)

Bases: object

__init__(*cfg*, *annfile*, *output_dir*)

eval(*results*, *epoch*)

class xmodaler.evaluation.**VCREvaler**(*cfg*, *annfile*, *output_dir*)

Bases: object

__init__(*cfg*, *annfile*, *output_dir*)

eval(*results_list*, *epoch*)

class xmodaler.evaluation.**RetrievalEvaler**(*cfg*, *annfile*, *output_dir*)

Bases: object

__init__(*cfg*, *annfile*, *output_dir*)

eval(*vfeats*, *tfeats*, *labels*)

3.6 xmodaler.functional

xmodaler.functional.**boxes_to_locfeats**(*boxes*, *image_w*, *image_h*)

xmodaler.functional.**caption_to_mask_tokens**(*caption*, *max_seq_length*, *tokenizer*, *need_g_tokens=False*, *need_no_mask_tokens=False*, *must_mask=False*)

xmodaler.functional.**decode_sequence**(*vocab*, *seq*)

xmodaler.functional.**decode_sequence_bert**(*tokenizer*, *seq*, *sep_token_id*)

xmodaler.functional.**dict_as_tensor**(*input_dict*)

xmodaler.functional.**dict_to_cuda**(*input_dict*)

xmodaler.functional.**expand_tensor**(*tensor*, *size*, *dim=1*)

xmodaler.functional.**clip_v_inputs**(*v_feats*, *spatials*, *image_mask*)

xmodaler.functional.**clip_t_inputs**(*input_txt*, *segment_ids*, *input_mask*)

xmodaler.functional.**iou**(*anchors*, *gt_boxes*)

anchors: (N, 4) ndarray of float gt_boxes: (K, 4) ndarray of float overlaps: (N, K) ndarray of overlap between boxes and query_boxes

xmodaler.functional.**load_vocab**(*path*)

xmodaler.functional.**pad_tensor**(*tensor*, *padding_value*, *use_mask*)

xmodaler.functional.**random_region**(*image_feats*, *overlaps*)

xmodaler.functional.**random_word**(*tokens*, *tokenizer*, *must_mask=False*)

xmodaler.functional.**read_lines**(*path*)

xmodaler.functional.**read_lines_set**(*path*)

xmodaler.functional.**read_np**(*path*)

xmodaler.functional.**read_np_bbox**(*path*, *max_feat_num*, *use_global_v=True*)

xmodaler.functional.**flat_list_of_lists**(*l*)

flatten a list of lists [[1,2], [3,4]] to [1,2,3,4]

3.7 xmodaler.losses

class xmodaler.losses.**BCEWithLogits**

Bases: Module

__init__()

Initializes internal Module state, shared by both nn.Module and ScriptModule.

classmethod add_config(*cfg*)

forward(*outputs_dict*)

Defines the computation performed at every call.

Should be overridden 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.

classmethod from_config(*cfg*)

training: bool

class xmodaler.losses.**BatchTriplet**(*margin*, *max_violation*)

Bases: Module

__init__(*margin*, *max_violation*)

Initializes internal Module state, shared by both nn.Module and ScriptModule.

forward(*outputs_dict*)

Defines the computation performed at every call.

Should be overridden 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.

classmethod from_config(*cfg*)

training: bool

class xmodaler.losses.**Triplet**(*margin*)

Bases: `Module`

__init__(*margin*)

Initializes internal Module state, shared by both nn.Module and ScriptModule.

classmethod add_config(*cfg*)

forward(*outputs_dict*)

Defines the computation performed at every call.

Should be overridden 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.

classmethod from_config(*cfg*)

training: bool

class xmodaler.losses.**CrossEntropy**

Bases: `Module`

__init__()

Initializes internal Module state, shared by both nn.Module and ScriptModule.

classmethod add_config(*cfg*)

forward(*outputs_dict*)

Defines the computation performed at every call.

Should be overridden 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.

classmethod from_config(*cfg*)

training: bool

class xmodaler.losses.**LabelSmoothing**(*, *label_smoothing*)

Bases: `Module`

Forward(*logits*, *targets*)

__init__(*, *label_smoothing*)

Initializes internal Module state, shared by both nn.Module and ScriptModule.

classmethod add_config(*cfg*)

forward(*outputs_dict*)

Defines the computation performed at every call.

Should be overridden 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.

classmethod from_config(*cfg*)

training: bool

class xmodaler.losses.**PretrainLosses**(*margin*, *max_violation*)

Bases: `Module`

__init__(*margin*, *max_violation*)

Initializes internal Module state, shared by both nn.Module and ScriptModule.

classmethod add_config(*cfg*)

forward(*batched_inputs*)

Defines the computation performed at every call.

Should be overridden 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.

classmethod from_config(*cfg*)

select_logits_targets_by_mask(*tensor*, *target*, *mask*)

training: bool

class xmodaler.losses.**RewardCriterion**(*eos_id*)

Bases: `Module`

__init__(*eos_id*)

Initializes internal Module state, shared by both nn.Module and ScriptModule.

classmethod add_config(*cfg*)

forward(*outputs_dict*)

Defines the computation performed at every call.

Should be overridden 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.

classmethod from_config(*cfg*)

training: bool

xmodaler.losses.**build_losses**(*cfg*)

xmodaler.losses.**build_rl_losses**(*cfg*)

3.8 xmodaler.lr_scheduler

xmodaler.lr_scheduler.**build_lr_scheduler**(*cfg*, *optimizer*, *data_size*)

class xmodaler.lr_scheduler.**StepLR**(*, *optimizer*, *step_size*, *gamma=0.1*)

Bases: `StepLR`

__init__(*, *optimizer*, *step_size*, *gamma=0.1*)

_get_closed_form_lr()

_initial_step()

Initialize step counts and performs a step

classmethod from_config(*cfg*, *optimizer*, *data_size*)

get_last_lr()

Return last computed learning rate by current scheduler.

get_lr()

load_state_dict(*state_dict*)

Loads the schedulers state.

Parameters

state_dict (*dict*) – scheduler state. Should be an object returned from a call to *state_dict()*.

print_lr(*is_verbose*, *group*, *lr*, *epoch=None*)

Display the current learning rate.

state_dict()

Returns the state of the scheduler as a `dict`.

It contains an entry for every variable in self.__dict__ which is not the optimizer.

step(*epoch=None*)

class xmodaler.lr_scheduler.**NoamLR**(*, *optimizer*, *model_size*, *factor*, *warmup*, *last_epoch=-1*)

Bases: `_LRScheduler`

__init__(*, *optimizer*, *model_size*, *factor*, *warmup*, *last_epoch=-1*)

_initial_step()

Initialize step counts and performs a step

classmethod from_config(*cfg*, *optimizer*, *data_size*)

get_last_lr()

Return last computed learning rate by current scheduler.

get_lr()

load_state_dict(*state_dict*)

Loads the schedulers state.

Parameters

state_dict (*dict*) – scheduler state. Should be an object returned from a call to *state_dict()*.

print_lr(*is_verbose*, *group*, *lr*, *epoch=None*)

Display the current learning rate.

state_dict()

Returns the state of the scheduler as a `dict`.

It contains an entry for every variable in self.__dict__ which is not the optimizer.

step(*epoch=None*)

class xmodaler.lr_scheduler.**WarmupConstant**(*, *optimizer*, *warmup_steps*, *last_epoch=-1*)

Bases: `LambdaLR`

Linear warmup and then constant. Linearly increases learning rate schedule from 0 to 1 over *warmup_steps* training steps. Keeps learning rate schedule equal to 1. after warmup_steps.

__init__(*, *optimizer*, *warmup_steps*, *last_epoch=-1*)

_initial_step()

Initialize step counts and performs a step

classmethod from_config(*cfg*, *optimizer*, *data_size*)

get_last_lr()

Return last computed learning rate by current scheduler.

get_lr()

load_state_dict(*state_dict*)

Loads the schedulers state.

When saving or loading the scheduler, please make sure to also save or load the state of the optimizer.

Parameters

state_dict (*dict*) – scheduler state. Should be an object returned from a call to *state_dict()*.

lr_lambda(*step*)

print_lr(*is_verbose*, *group*, *lr*, *epoch=None*)

Display the current learning rate.

state_dict()

Returns the state of the scheduler as a `dict`.

It contains an entry for every variable in self.__dict__ which is not the optimizer. The learning rate lambda functions will only be saved if they are callable objects and not if they are functions or lambdas.

When saving or loading the scheduler, please make sure to also save or load the state of the optimizer.

step(*epoch=None*)

class xmodaler.lr_scheduler.**WarmupLinear**(*, *optimizer*, *min_lr*, *warmup_steps*, *t_total*, *last_epoch=-1*)

Bases: `LambdaLR`

Linear warmup and then linear decay. Linearly increases learning rate from 0 to 1 over *warmup_steps* training steps. Linearly decreases learning rate from 1. to 0. over remaining *t_total - warmup_steps* steps.

__init__(*, *optimizer*, *min_lr*, *warmup_steps*, *t_total*, *last_epoch=-1*)

_initial_step()

Initialize step counts and performs a step

classmethod from_config(*cfg*, *optimizer*, *data_size*)

get_last_lr()

Return last computed learning rate by current scheduler.

get_lr()

load_state_dict(*state_dict*)

Loads the schedulers state.

When saving or loading the scheduler, please make sure to also save or load the state of the optimizer.

Parameters

state_dict (*dict*) – scheduler state. Should be an object returned from a call to *state_dict()*.

lr_lambda(*step*)

print_lr(*is_verbose*, *group*, *lr*, *epoch=None*)

Display the current learning rate.

state_dict()

Returns the state of the scheduler as a `dict`.

It contains an entry for every variable in self.__dict__ which is not the optimizer. The learning rate lambda functions will only be saved if they are callable objects and not if they are functions or lambdas.

When saving or loading the scheduler, please make sure to also save or load the state of the optimizer.

step(*epoch=None*)

class xmodaler.lr_scheduler.**WarmupCosine**(*, *optimizer*, *min_lr*, *warmup_steps*, *t_total*, *cycles=0.5*, *last_epoch=-1*)

Bases: `LambdaLR`

Linear warmup and then cosine decay. Linearly increases learning rate from 0 to 1 over *warmup_steps* training steps. Decreases learning rate from 1. to 0. over remaining *t_total - warmup_steps* steps following a cosine curve. If *cycles* (default=0.5) is different from default, learning rate follows cosine function after warmup.

__init__(*, *optimizer*, *min_lr*, *warmup_steps*, *t_total*, *cycles=0.5*, *last_epoch=-1*)

_initial_step()

Initialize step counts and performs a step

classmethod from_config(*cfg*, *optimizer*, *data_size*)

get_last_lr()

Return last computed learning rate by current scheduler.

get_lr()

load_state_dict(*state_dict*)

Loads the schedulers state.

When saving or loading the scheduler, please make sure to also save or load the state of the optimizer.

Parameters

state_dict (*dict*) – scheduler state. Should be an object returned from a call to *state_dict()*.

lr_lambda(*step*)

print_lr(*is_verbose*, *group*, *lr*, *epoch=None*)

Display the current learning rate.

state_dict()

Returns the state of the scheduler as a `dict`.

It contains an entry for every variable in self.__dict__ which is not the optimizer. The learning rate lambda functions will only be saved if they are callable objects and not if they are functions or lambdas.

When saving or loading the scheduler, please make sure to also save or load the state of the optimizer.

step(*epoch=None*)

class xmodaler.lr_scheduler.**WarmupCosineWithHardRestarts**(*, *optimizer*, *warmup_steps*, *t_total*, *cycles=1.0*, *last_epoch=-1*)

Bases: `LambdaLR`

Linear warmup and then cosine cycles with hard restarts. Linearly increases learning rate from 0 to 1 over *warmup_steps* training steps. If *cycles* (default=1.) is different from default, learning rate follows *cycles* times a cosine decaying learning rate (with hard restarts).

__init__(*, *optimizer*, *warmup_steps*, *t_total*, *cycles=1.0*, *last_epoch=-1*)

_initial_step()

Initialize step counts and performs a step

classmethod from_config(*cfg*, *optimizer*, *data_size*)

get_last_lr()

Return last computed learning rate by current scheduler.

get_lr()

load_state_dict(*state_dict*)

Loads the schedulers state.

When saving or loading the scheduler, please make sure to also save or load the state of the optimizer.

Parameters

state_dict (*dict*) – scheduler state. Should be an object returned from a call to *state_dict()*.

lr_lambda(*step*)

print_lr(*is_verbose*, *group*, *lr*, *epoch=None*)

Display the current learning rate.

state_dict()

Returns the state of the scheduler as a `dict`.

It contains an entry for every variable in self.__dict__ which is not the optimizer. The learning rate lambda functions will only be saved if they are callable objects and not if they are functions or lambdas.

When saving or loading the scheduler, please make sure to also save or load the state of the optimizer.

step(*epoch=None*)

class xmodaler.lr_scheduler.**WarmupMultiStepLR**(*, *optimizer*, *milestones*, *gamma=0.1*, *warmup_factor=0.3333333333333333*, *warmup_iters=500*, *warmup_method='linear'*, *last_epoch=-1*)

Bases: `_LRScheduler`

__init__(*, *optimizer*, *milestones*, *gamma=0.1*, *warmup_factor=0.3333333333333333*, *warmup_iters=500*, *warmup_method='linear'*, *last_epoch=-1*)

_initial_step()

Initialize step counts and performs a step

classmethod from_config(*cfg*, *optimizer*, *data_size*)

get_last_lr()

Return last computed learning rate by current scheduler.

get_lr()

load_state_dict(*state_dict*)

Loads the schedulers state.

Parameters

state_dict (*dict*) – scheduler state. Should be an object returned from a call to *state_dict()*.

print_lr(*is_verbose*, *group*, *lr*, *epoch=None*)

Display the current learning rate.

state_dict()

Returns the state of the scheduler as a `dict`.

It contains an entry for every variable in self.__dict__ which is not the optimizer.

step(*epoch=None*)

class xmodaler.lr_scheduler.**MultiStepLR**(*, *optimizer*, *milestones*, *gamma=0.1*, *last_epoch=-1*)

Bases: `MultiStepLR`

__init__(*, *optimizer*, *milestones*, *gamma=0.1*, *last_epoch=-1*)

_get_closed_form_lr()

_initial_step()

Initialize step counts and performs a step

classmethod from_config(*cfg*, *optimizer*, *data_size*)

get_last_lr()

Return last computed learning rate by current scheduler.

get_lr()

load_state_dict(*state_dict*)

Loads the schedulers state.

Parameters

state_dict (*dict*) – scheduler state. Should be an object returned from a call to *state_dict()*.

print_lr(*is_verbose*, *group*, *lr*, *epoch=None*)

Display the current learning rate.

state_dict()

Returns the state of the scheduler as a `dict`.

It contains an entry for every variable in self.__dict__ which is not the optimizer.

step(*epoch=None*)

class xmodaler.lr_scheduler.**FixLR**(*, *optimizer*, *last_epoch=-1*)

Bases: `LambdaLR`

Fix LR

__init__(*, *optimizer*, *last_epoch=-1*)

_initial_step()

Initialize step counts and performs a step

classmethod from_config(*cfg*, *optimizer*, *data_size*)

get_last_lr()

Return last computed learning rate by current scheduler.

get_lr()

load_state_dict(*state_dict*)

Loads the schedulers state.

When saving or loading the scheduler, please make sure to also save or load the state of the optimizer.

Parameters

state_dict (*dict*) – scheduler state. Should be an object returned from a call to *state_dict()*.

lr_lambda(*step*)

print_lr(*is_verbose*, *group*, *lr*, *epoch=None*)

Display the current learning rate.

state_dict()

Returns the state of the scheduler as a `dict`.

It contains an entry for every variable in self.__dict__ which is not the optimizer. The learning rate lambda functions will only be saved if they are callable objects and not if they are functions or lambdas.

When saving or loading the scheduler, please make sure to also save or load the state of the optimizer.

step(*epoch=None*)

3.9 xmodaler.modeling

3.9.1 xmodaler.modeling.decode_strategy

xmodaler.modeling.decode_strategy.**build_beam_searcher**(*cfg*)

xmodaler.modeling.decode_strategy.**build_greedy_decoder**(*cfg*)

class xmodaler.modeling.decode_strategy.**GreedyDecoder**(*, *vocab_path*, *beam_size*, *max_seq_len*, *bert_tokenizer*, *bos_token_id*, *eos_token_id*)

Bases: *DecodeStrategy*

_abc_impl = <_abc_data object>

_backward_hooks: Dict[int, Callable]

_buffers: Dict[str, Optional[Tensor]]

_forward(*batched_inputs*, *model*)

_forward_hooks: Dict[int, Callable]

_forward_pre_hooks: Dict[int, Callable]

_is_full_backward_hook: Optional[bool]

_load_state_dict_post_hooks: Dict[int, Callable]

_load_state_dict_pre_hooks: Dict[int, Callable]

_modules: Dict[str, Optional[Module]]

_non_persistent_buffers_set: Set[str]

_parameters: Dict[str, Optional[Parameter]]

_state_dict_hooks: Dict[int, Callable]

training: bool

class xmodaler.modeling.decode_strategy.**BeamSearcher**(*, *vocab_path*, *beam_size*, *max_seq_len*, *bert_tokenizer*, *bos_token_id*, *eos_token_id*)

Bases: *DecodeStrategy*

_abc_impl = <_abc_data object>

_backward_hooks: Dict[int, Callable]

_buffers: Dict[str, Optional[Tensor]]

_expand_state(*states*, *selected_beam*, *batch_size*, *beam_size*, *cur_beam_size*)

_forward(*batched_inputs*, *model*)

_forward_hooks: Dict[int, Callable]

_forward_pre_hooks: Dict[int, Callable]

_is_full_backward_hook: Optional[bool]

_load_state_dict_post_hooks: Dict[int, Callable]

_load_state_dict_pre_hooks: Dict[int, Callable]

_modules: Dict[str, Optional[Module]]

_non_persistent_buffers_set: Set[str]

_parameters: Dict[str, Optional[Parameter]]

_select(*batch_size*, *beam_size*, *t*, *candidate_logprob*)

_state_dict_hooks: Dict[int, Callable]

training: bool

class xmodaler.modeling.decode_strategy.decode_strategy.**DecodeStrategy**(*, *vocab_path*, *beam_size*, *max_seq_len*, *bert_tokenizer*, *bos_token_id*, *eos_token_id*)

Bases: Module

__init__(*, *vocab_path*, *beam_size*, *max_seq_len*, *bert_tokenizer*, *bos_token_id*, *eos_token_id*)

Initializes internal Module state, shared by both nn.Module and ScriptModule.

_abc_impl = <_abc_data object>

_backward_hooks: Dict[int, Callable]

_buffers: Dict[str, Optional[Tensor]]

abstract _forward(*batched_inputs*, *model*)

_forward_hooks: Dict[int, Callable]

_forward_pre_hooks: Dict[int, Callable]

_is_full_backward_hook: Optional[bool]

_load_state_dict_post_hooks: Dict[int, Callable]

_load_state_dict_pre_hooks: Dict[int, Callable]

_modules: Dict[str, Optional[Module]]

_non_persistent_buffers_set: Set[str]

_parameters: Dict[str, Optional[Parameter]]

_state_dict_hooks: Dict[int, Callable]

forward(*batched_inputs*, *output_sents*, *model*)

Defines the computation performed at every call.

Should be overridden 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.

classmethod from_config(*cfg*)

training: bool

3.9.2 xmodaler.modeling.decoder

xmodaler.modeling.decoder.**build_decoder**(*cfg*)

xmodaler.modeling.decoder.**add_decoder_config**(*cfg*, *tmp_cfg*)

class xmodaler.modeling.decoder.**UpDownDecoder**(*, *hidden_size: int*, *token_embed_dim: int*, *visual_embed_dim: int*, *att_embed_size: int*, *dropout1: float*, *dropout2: float*, *att_embed_dropout: float*)

Bases: `Decoder`

__init__(*, *hidden_size: int*, *token_embed_dim: int*, *visual_embed_dim: int*, *att_embed_size: int*, *dropout1: float*, *dropout2: float*, *att_embed_dropout: float*)

Initializes internal Module state, shared by both nn.Module and ScriptModule.

_abc_impl = <_abc_data object>

_backward_hooks: Dict[int, Callable]

_buffers: Dict[str, Optional[Tensor]]

_forward_hooks: Dict[int, Callable]

_forward_pre_hooks: Dict[int, Callable]

_is_full_backward_hook: Optional[bool]

_load_state_dict_post_hooks: Dict[int, Callable]

_load_state_dict_pre_hooks: Dict[int, Callable]

_modules: Dict[str, Optional[Module]]

_non_persistent_buffers_set: Set[str]

_parameters: Dict[str, Optional[Parameter]]

_state_dict_hooks: Dict[int, Callable]

classmethod add_config(*cfg*)

forward(*batched_inputs*)

Defines the computation performed at every call.

Should be overridden 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.

classmethod from_config(*cfg*)

preprocess(*batched_inputs*)

training: bool

class xmodaler.modeling.decoder.**SALSTMDecoder**(*, *hidden_size: int*, *token_embed_dim: int*, *visual_embed_dim: int*, *att_embed_size: int*, *att_embed_dropout: float*)

Bases: `Decoder`

__init__(*, *hidden_size: int*, *token_embed_dim: int*, *visual_embed_dim: int*, *att_embed_size: int*, *att_embed_dropout: float*)

Initializes internal Module state, shared by both nn.Module and ScriptModule.

_abc_impl = <_abc_data object>

_backward_hooks: Dict[int, Callable]

_buffers: Dict[str, Optional[Tensor]]

_forward_hooks: Dict[int, Callable]

_forward_pre_hooks: Dict[int, Callable]

_is_full_backward_hook: Optional[bool]

_load_state_dict_post_hooks: Dict[int, Callable]

_load_state_dict_pre_hooks: Dict[int, Callable]

_modules: Dict[str, Optional[Module]]

_non_persistent_buffers_set: Set[str]

_parameters: Dict[str, Optional[Parameter]]

_state_dict_hooks: Dict[int, Callable]

classmethod add_config(*cfg*)

forward(*batched_inputs*)

Defines the computation performed at every call.

Should be overridden 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.

classmethod from_config(*cfg*)

preprocess(*batched_inputs*)

training: bool

class xmodaler.modeling.decoder.**MPLSTMDecoder**(*, *hidden_size: int*, *token_embed_dim: int*)

Bases: Decoder

__init__(*, *hidden_size: int*, *token_embed_dim: int*)

Initializes internal Module state, shared by both nn.Module and ScriptModule.

_abc_impl = <_abc_data object>

_backward_hooks: Dict[int, Callable]

_buffers: Dict[str, Optional[Tensor]]

_forward_hooks: Dict[int, Callable]

_forward_pre_hooks: Dict[int, Callable]

_is_full_backward_hook: Optional[bool]

_load_state_dict_post_hooks: Dict[int, Callable]

_load_state_dict_pre_hooks: Dict[int, Callable]

_modules: Dict[str, Optional[Module]]

_non_persistent_buffers_set: Set[str]

_parameters: Dict[str, Optional[Parameter]]

_state_dict_hooks: Dict[int, Callable]

classmethod add_config(*cfg*)

forward(*batched_inputs*)

Defines the computation performed at every call.

Should be overridden 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.

---

classmethod from_config(*cfg*)

preprocess(*batched_inputs*)

training: bool

class xmodaler.modeling.decoder.**TransformerDecoder**(*, *num_generation_layers: int*, *bert_generation_layers*)

Bases: Decoder

__init__(*, *num_generation_layers: int*, *bert_generation_layers*)

Initializes internal Module state, shared by both nn.Module and ScriptModule.

_abc_impl = <_abc_data object>

_backward_hooks: Dict[int, Callable]

_buffers: Dict[str, Optional[Tensor]]

_forward_hooks: Dict[int, Callable]

_forward_pre_hooks: Dict[int, Callable]

_is_full_backward_hook: Optional[bool]

_load_state_dict_post_hooks: Dict[int, Callable]

_load_state_dict_pre_hooks: Dict[int, Callable]

_modules: Dict[str, Optional[Module]]

_non_persistent_buffers_set: Set[str]

_parameters: Dict[str, Optional[Parameter]]

_state_dict_hooks: Dict[int, Callable]

classmethod add_config(*cfg*)

forward(*batched_inputs*)

Defines the computation performed at every call.

Should be overridden 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.

---

classmethod from_config(*cfg*)

training: bool

class xmodaler.modeling.decoder.**DecoupleBertDecoder**(*, *num_understanding_layers: int*, *num_generation_layers: int*, *u_layer_drop: float*, *g_layer_drop: float*, *bert_understanding_layers*, *bert_generation_layers*)

Bases: `Decoder`

__init__(*, *num_understanding_layers: int*, *num_generation_layers: int*, *u_layer_drop: float*, *g_layer_drop: float*, *bert_understanding_layers*, *bert_generation_layers*)

Initializes internal Module state, shared by both nn.Module and ScriptModule.

_abc_impl = <_abc_data object>

_backward_hooks: Dict[int, Callable]

_buffers: Dict[str, Optional[Tensor]]

_forward_hooks: Dict[int, Callable]

_forward_pre_hooks: Dict[int, Callable]

_is_full_backward_hook: Optional[bool]

_load_state_dict_post_hooks: Dict[int, Callable]

_load_state_dict_pre_hooks: Dict[int, Callable]

_modules: Dict[str, Optional[Module]]

_non_persistent_buffers_set: Set[str]

_parameters: Dict[str, Optional[Parameter]]

_state_dict_hooks: Dict[int, Callable]

classmethod add_config(*cfg*)

forward(*batched_inputs*)

Defines the computation performed at every call.

Should be overridden 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.

---

classmethod from_config(*cfg*)

training: bool

class xmodaler.modeling.decoder.**MeshedDecoder**(*, *d_model: int*, *num_layer: int*, *num_att_head: int*, *d_ff: int*, *dropout: float*, *padding_idx: int*, *enc_layer_num: int*)

Bases: `Decoder`

__init__(*, *d_model: int*, *num_layer: int*, *num_att_head: int*, *d_ff: int*, *dropout: float*, *padding_idx: int*, *enc_layer_num: int*)

Initializes internal Module state, shared by both nn.Module and ScriptModule.

_abc_impl = <_abc_data object>

_backward_hooks: Dict[int, Callable]

_buffers: Dict[str, Optional[Tensor]]

_forward_hooks: Dict[int, Callable]

_forward_pre_hooks: Dict[int, Callable]

_is_full_backward_hook: Optional[bool]

_load_state_dict_post_hooks: Dict[int, Callable]

_load_state_dict_pre_hooks: Dict[int, Callable]

_modules: Dict[str, Optional[Module]]

_non_persistent_buffers_set: Set[str]

_parameters: Dict[str, Optional[Parameter]]

_state_dict_hooks: Dict[int, Callable]

classmethod add_config(*cfg*)

forward(*batched_inputs*)

Defines the computation performed at every call.

Should be overridden 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.

---

classmethod from_config(*cfg*)

training: bool

class xmodaler.modeling.decoder.**XLANDecoder**(*, *hidden_size: int*, *ctx_drop: float*, *bilinear_dim: int*, *att_heads: int*, *att_mid_dim: int*, *att_mid_drop: float*, *att_embed_dropout: float*, *layer_num: int*, *act_type: str*, *elu_alpha: float*)

Bases: `Decoder`

__init__(*, *hidden_size: int*, *ctx_drop: float*, *bilinear_dim: int*, *att_heads: int*, *att_mid_dim: int*, *att_mid_drop: float*, *att_embed_dropout: float*, *layer_num: int*, *act_type: str*, *elu_alpha: float*)

Initializes internal Module state, shared by both nn.Module and ScriptModule.

_abc_impl = <_abc_data object>

_backward_hooks: Dict[int, Callable]

_buffers: Dict[str, Optional[Tensor]]

_forward_hooks: Dict[int, Callable]

_forward_pre_hooks: Dict[int, Callable]

_is_full_backward_hook: Optional[bool]

_load_state_dict_post_hooks: Dict[int, Callable]

_load_state_dict_pre_hooks: Dict[int, Callable]

_modules: Dict[str, Optional[Module]]

_non_persistent_buffers_set: Set[str]

_parameters: Dict[str, Optional[Parameter]]

_state_dict_hooks: Dict[int, Callable]

classmethod add_config(*cfg*)

forward(*batched_inputs*)

Defines the computation performed at every call.

Should be overridden 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.

classmethod from_config(*cfg*)

preprocess(*batched_inputs*)

training: bool

class xmodaler.modeling.decoder.**TDConvEDDecoder**(*, *num_hidden_layers: int*, *hidden_size: int*, *kernel_sizes: list*, *conv_dropout: float*, *att_embed_size: int*, *att_embed_dropout: float*, *use_norm: bool*)

Bases: `Module`

__init__(*, *num_hidden_layers: int*, *hidden_size: int*, *kernel_sizes: list*, *conv_dropout: float*, *att_embed_size: int*, *att_embed_dropout: float*, *use_norm: bool*)

Initializes internal Module state, shared by both nn.Module and ScriptModule.

_backward_hooks: Dict[int, Callable]

_buffers: Dict[str, Optional[Tensor]]

_clear_decoding_buffer()

_forward_hooks: Dict[int, Callable]

_forward_pre_hooks: Dict[int, Callable]

_init_decoding_buffer(*batch_size*)

_is_full_backward_hook: Optional[bool]

_load_state_dict_post_hooks: Dict[int, Callable]

_load_state_dict_pre_hooks: Dict[int, Callable]

_modules: Dict[str, Optional[Module]]

_non_persistent_buffers_set: Set[str]

_parameters: Dict[str, Optional[Parameter]]

_state_dict_hooks: Dict[int, Callable]

classmethod add_config(*cfg*)

forward(*batched_inputs*)

Defines the computation performed at every call.

Should be overridden 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.

classmethod from_config(*cfg*)

preprocess(*batched_inputs*)

training: bool

class xmodaler.modeling.decoder.**AttributeDecoder**(*, *hidden_size: int*, *token_embed_dim: int*, *visual_feat_dim: int*, *attribute_dim: int*, *dropout: float*)

Bases: `Decoder`

__init__(*, *hidden_size: int*, *token_embed_dim: int*, *visual_feat_dim: int*, *attribute_dim: int*, *dropout: float*)

Initializes internal Module state, shared by both nn.Module and ScriptModule.

_abc_impl = <_abc_data object>

_backward_hooks: Dict[int, Callable]

_buffers: Dict[str, Optional[Tensor]]

_forward_hooks: Dict[int, Callable]

_forward_pre_hooks: Dict[int, Callable]

_is_full_backward_hook: Optional[bool]

_load_state_dict_post_hooks: Dict[int, Callable]

_load_state_dict_pre_hooks: Dict[int, Callable]

_modules: Dict[str, Optional[Module]]

_non_persistent_buffers_set: Set[str]

_parameters: Dict[str, Optional[Parameter]]

_state_dict_hooks: Dict[int, Callable]

classmethod add_config(*cfg*)

forward(*batched_inputs*)

Defines the computation performed at every call.

Should be overridden 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.

classmethod from_config(*cfg*)

preprocess(*batched_inputs*)

training: bool

3.9.3 xmodaler.modeling.embedding

xmodaler.modeling.embedding.**build_embeddings**(*cfg*, *name*)

class xmodaler.modeling.embedding.**TokenBaseEmbedding**(*, *dim: int*, *vocab_size: int*, ***kwargs*)

Bases: Module

__init__(*, *dim: int*, *vocab_size: int*, ***kwargs*)

Initializes internal Module state, shared by both nn.Module and ScriptModule.

_backward_hooks: Dict[int, Callable]

_buffers: Dict[str, Optional[Tensor]]

_forward(*input_ids*, *token_type_ids=None*, *time_step=None*)

_forward_hooks: Dict[int, Callable]

_forward_pre_hooks: Dict[int, Callable]

_is_full_backward_hook: Optional[bool]

_load_state_dict_post_hooks: Dict[int, Callable]

_load_state_dict_pre_hooks: Dict[int, Callable]

_modules: Dict[str, Optional[Module]]

_non_persistent_buffers_set: Set[str]

_parameters: Dict[str, Optional[Parameter]]

_state_dict_hooks: Dict[int, Callable]

forward(*batched_inputs*)

Defines the computation performed at every call.

Should be overridden 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.

classmethod from_config(*cfg*)

training: bool

class xmodaler.modeling.embedding.**VisualBaseEmbedding**(*, *in_dim: int*, *out_dim: int*, ***kwargs*)

Bases: Module

__init__(*, *in_dim: int*, *out_dim: int*, ***kwargs*)

Initializes internal Module state, shared by both nn.Module and ScriptModule.

_backward_hooks: Dict[int, Callable]

_buffers: Dict[str, Optional[Tensor]]

_forward_hooks: Dict[int, Callable]

_forward_pre_hooks: Dict[int, Callable]

_is_full_backward_hook: Optional[bool]

_load_state_dict_post_hooks: Dict[int, Callable]

_load_state_dict_pre_hooks: Dict[int, Callable]

_modules: Dict[str, Optional[Module]]

_non_persistent_buffers_set: Set[str]

_parameters: Dict[str, Optional[Parameter]]

_state_dict_hooks: Dict[int, Callable]

forward(*batched_inputs*)

Defines the computation performed at every call.

Should be overridden 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.

---

classmethod from_config(*cfg*)

training: bool

class xmodaler.modeling.embedding.**VisualIdentityEmbedding**

Bases: `Module`

__init__()

Initializes internal Module state, shared by both nn.Module and ScriptModule.

_backward_hooks: Dict[int, Callable]

_buffers: Dict[str, Optional[Tensor]]

_forward_hooks: Dict[int, Callable]

_forward_pre_hooks: Dict[int, Callable]

_is_full_backward_hook: Optional[bool]

_load_state_dict_post_hooks: Dict[int, Callable]

_load_state_dict_pre_hooks: Dict[int, Callable]

_modules: Dict[str, Optional[Module]]

_non_persistent_buffers_set: Set[str]

_parameters: Dict[str, Optional[Parameter]]

_state_dict_hooks: Dict[int, Callable]

forward(*batched_inputs*)

Defines the computation performed at every call.

Should be overridden 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.

---

classmethod from_config(*cfg*)

training: bool

class xmodaler.modeling.embedding.**TDConvEDVisualBaseEmbedding**(*, *in_dim: int*, *out_dim: int*, ***kwargs*)

Bases: `Module`

__init__(*, *in_dim: int*, *out_dim: int*, ***kwargs*)

Initializes internal Module state, shared by both nn.Module and ScriptModule.

_backward_hooks: Dict[int, Callable]

_buffers: Dict[str, Optional[Tensor]]

_forward_hooks: Dict[int, Callable]

_forward_pre_hooks: Dict[int, Callable]

_is_full_backward_hook: Optional[bool]

_load_state_dict_post_hooks: Dict[int, Callable]

_load_state_dict_pre_hooks: Dict[int, Callable]

_modules: Dict[str, Optional[Module]]

_non_persistent_buffers_set: Set[str]

_parameters: Dict[str, Optional[Parameter]]

_state_dict_hooks: Dict[int, Callable]

forward(*batched_inputs*)

Defines the computation performed at every call.

Should be overridden 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.

---

classmethod from_config(*cfg*)

training: bool

class xmodaler.modeling.embedding.position_embedding.**SinusoidEncoding**(*dim*, *max_len*)

Bases: Module

__init__(*dim*, *max_len*)

Initializes internal Module state, shared by both nn.Module and ScriptModule.

_backward_hooks: Dict[int, Callable]

_buffers: Dict[str, Optional[Tensor]]

_forward_hooks: Dict[int, Callable]

_forward_pre_hooks: Dict[int, Callable]

_is_full_backward_hook: Optional[bool]

_load_state_dict_post_hooks: Dict[int, Callable]

_load_state_dict_pre_hooks: Dict[int, Callable]

_modules: Dict[str, Optional[Module]]

_non_persistent_buffers_set: Set[str]

_parameters: Dict[str, Optional[Parameter]]

_state_dict_hooks: Dict[int, Callable]

forward(*x*)

Defines the computation performed at every call.

Should be overridden 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.

---

training: bool

class xmodaler.modeling.embedding.position_embedding.**NNEmbeddingEncoding**(*dim*, *max_len*)

Bases: Module

__init__(*dim*, *max_len*)

Initializes internal Module state, shared by both nn.Module and ScriptModule.

_backward_hooks: Dict[int, Callable]

_buffers: Dict[str, Optional[Tensor]]

_forward_hooks: Dict[int, Callable]

_forward_pre_hooks: Dict[int, Callable]

_is_full_backward_hook: Optional[bool]

_load_state_dict_post_hooks: Dict[int, Callable]

_load_state_dict_pre_hooks: Dict[int, Callable]

_modules: Dict[str, Optional[Module]]

_non_persistent_buffers_set: Set[str]

_parameters: Dict[str, Optional[Parameter]]

_state_dict_hooks: Dict[int, Callable]

forward(*x*)

Defines the computation performed at every call.

Should be overridden 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.

training: bool

3.9.4 xmodaler.modeling.encoder

xmodaler.modeling.encoder.**build_encoder**(*cfg*)

xmodaler.modeling.encoder.**add_encoder_config**(*cfg*, *tmp_cfg*)

class xmodaler.modeling.encoder.**Encoder**

Bases: `Module`

__init__()

Initializes internal Module state, shared by both nn.Module and ScriptModule.

_backward_hooks: Dict[int, Callable]

_buffers: Dict[str, Optional[Tensor]]

_forward_hooks: Dict[int, Callable]

_forward_pre_hooks: Dict[int, Callable]

_is_full_backward_hook: Optional[bool]

_load_state_dict_post_hooks: Dict[int, Callable]

_load_state_dict_pre_hooks: Dict[int, Callable]

_modules: Dict[str, Optional[Module]]

_non_persistent_buffers_set: Set[str]

_parameters: Dict[str, Optional[Parameter]]

_state_dict_hooks: Dict[int, Callable]

classmethod add_config(*cfg*)

forward(*batched_inputs*, *mode=None*)

Defines the computation performed at every call.

Should be overridden 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.

classmethod from_config(*cfg*)

training: bool

class xmodaler.modeling.encoder.**UpDownEncoder**

Bases: `Module`

__init__()

Initializes internal Module state, shared by both nn.Module and ScriptModule.

_backward_hooks: Dict[int, Callable]

_buffers: Dict[str, Optional[Tensor]]

_forward_hooks: Dict[int, Callable]

_forward_pre_hooks: Dict[int, Callable]

_is_full_backward_hook: Optional[bool]

_load_state_dict_post_hooks: Dict[int, Callable]

_load_state_dict_pre_hooks: Dict[int, Callable]

_modules: Dict[str, Optional[Module]]

_non_persistent_buffers_set: Set[str]

_parameters: Dict[str, Optional[Parameter]]

_state_dict_hooks: Dict[int, Callable]

classmethod add_config(*cfg*)

forward(*batched_inputs*, *mode=None*)

Defines the computation performed at every call.

Should be overridden 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.

classmethod from_config(*cfg*)

training: bool

class xmodaler.modeling.encoder.**TransformerEncoder**(*, *num_hidden_layers: int*, *bert_layers*)

Bases: Module

__init__(*, *num_hidden_layers: int*, *bert_layers*)

Initializes internal Module state, shared by both nn.Module and ScriptModule.

_backward_hooks: Dict[int, Callable]

_buffers: Dict[str, Optional[Tensor]]

_forward_hooks: Dict[int, Callable]

_forward_pre_hooks: Dict[int, Callable]

_is_full_backward_hook: Optional[bool]

_load_state_dict_post_hooks: Dict[int, Callable]

_load_state_dict_pre_hooks: Dict[int, Callable]

_modules: Dict[str, Optional[Module]]

_non_persistent_buffers_set: Set[str]

_parameters: Dict[str, Optional[Parameter]]

_state_dict_hooks: Dict[int, Callable]

classmethod add_config(*cfg*)

forward(*batched_inputs*, *mode=None*)

Defines the computation performed at every call.

Should be overridden 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.

---

classmethod from_config(*cfg*)

training: bool

class xmodaler.modeling.encoder.**SingleStreamBertEncoder**(*, *num_hidden_layers: int*, *bert_layers*)

Bases: Module

__init__(*, *num_hidden_layers: int*, *bert_layers*)

Initializes internal Module state, shared by both nn.Module and ScriptModule.

_backward_hooks: Dict[int, Callable]

_buffers: Dict[str, Optional[Tensor]]

_forward_hooks: Dict[int, Callable]

_forward_pre_hooks: Dict[int, Callable]

_is_full_backward_hook: Optional[bool]

_load_state_dict_post_hooks: Dict[int, Callable]

_load_state_dict_pre_hooks: Dict[int, Callable]

_modules: Dict[str, Optional[Module]]

_non_persistent_buffers_set: Set[str]

_parameters: Dict[str, Optional[Parameter]]

_state_dict_hooks: Dict[int, Callable]

classmethod add_config(*cfg*)

forward(*batched_inputs*, *mode=None*)

Defines the computation performed at every call.

Should be overridden 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.

---

classmethod from_config(*cfg*)

training: bool

class xmodaler.modeling.encoder.**TwoStreamBertEncoder**(*, *num_hidden_layers: int*, *v_num_hidden_layers: int*, *layer_drop: float*, *v_layer_drop: float*, *bert_layers*, *v_bert_layers*)

Bases: `Module`

__init__(*, *num_hidden_layers: int*, *v_num_hidden_layers: int*, *layer_drop: float*, *v_layer_drop: float*, *bert_layers*, *v_bert_layers*)

Initializes internal Module state, shared by both nn.Module and ScriptModule.

_backward_hooks: Dict[int, Callable]

_buffers: Dict[str, Optional[Tensor]]

_forward_hooks: Dict[int, Callable]

_forward_pre_hooks: Dict[int, Callable]

_is_full_backward_hook: Optional[bool]

_load_state_dict_post_hooks: Dict[int, Callable]

_load_state_dict_pre_hooks: Dict[int, Callable]

_modules: Dict[str, Optional[Module]]

_non_persistent_buffers_set: Set[str]

_parameters: Dict[str, Optional[Parameter]]

_state_dict_hooks: Dict[int, Callable]

classmethod add_config(*cfg*)

forward(*batched_inputs*, *mode=None*)

Defines the computation performed at every call.

Should be overridden 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.

classmethod from_config(*cfg*)

training: bool

class xmodaler.modeling.encoder.**GCNEncoder**(*, *in_dim: int*, *out_dim: int*, *relation_num: int*, *dropout: float*)

Bases: `Module`

__init__(*, *in_dim: int*, *out_dim: int*, *relation_num: int*, *dropout: float*)

Initializes internal Module state, shared by both nn.Module and ScriptModule.

_backward_hooks: Dict[int, Callable]

_buffers: Dict[str, Optional[Tensor]]

_forward_hooks: Dict[int, Callable]

_forward_pre_hooks: Dict[int, Callable]

_is_full_backward_hook: Optional[bool]

_load_state_dict_post_hooks: Dict[int, Callable]

_load_state_dict_pre_hooks: Dict[int, Callable]

_modules: Dict[str, Optional[Module]]

_non_persistent_buffers_set: Set[str]

_parameters: Dict[str, Optional[Parameter]]

_state_dict_hooks: Dict[int, Callable]

classmethod add_config(*cfg*)

forward(*batched_inputs*, *mode=None*)

Defines the computation performed at every call.

Should be overridden 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.

classmethod from_config(*cfg*)

training: bool

class xmodaler.modeling.encoder.**MemoryAugmentedEncoder**(*, *input_dim: int*, *d_model: int*, *num_layer: int*, *num_att_head: int*, *num_att_memory: int*, *d_ff: int*, *dropout: float*, *padding_idx: int*)

Bases: Module

__init__(*, *input_dim: int*, *d_model: int*, *num_layer: int*, *num_att_head: int*, *num_att_memory: int*, *d_ff: int*, *dropout: float*, *padding_idx: int*)

Initializes internal Module state, shared by both nn.Module and ScriptModule.

_backward_hooks: Dict[int, Callable]

_buffers: Dict[str, Optional[Tensor]]

_forward_hooks: Dict[int, Callable]

_forward_pre_hooks: Dict[int, Callable]

_get_global_feat(*feats*, *masks*)

_is_full_backward_hook: Optional[bool]

_load_state_dict_post_hooks: Dict[int, Callable]

_load_state_dict_pre_hooks: Dict[int, Callable]

_modules: Dict[str, Optional[Module]]

_non_persistent_buffers_set: Set[str]

_parameters: Dict[str, Optional[Parameter]]

_state_dict_hooks: Dict[int, Callable]

classmethod add_config(*cfg*)

forward(*batched_inputs*, *mode=None*)

Defines the computation performed at every call.

Should be overridden 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.

---

classmethod from_config(*cfg*)

training: bool

class xmodaler.modeling.encoder.**LowRankBilinearEncoder**(*, *embed_dim: int*, *att_heads: int*, *att_mid_dim: int*, *att_mid_drop: float*, *dropout: float*, *bifeat_emb_dropout: float*, *layer_num: int*, *emb_act_type: str*, *act_type: str*, *elu_alpha: float*)

Bases: `Module`

__init__(*, *embed_dim: int*, *att_heads: int*, *att_mid_dim: int*, *att_mid_drop: float*, *dropout: float*, *bifeat_emb_dropout: float*, *layer_num: int*, *emb_act_type: str*, *act_type: str*, *elu_alpha: float*)

Initializes internal Module state, shared by both nn.Module and ScriptModule.

_backward_hooks: Dict[int, Callable]

_buffers: Dict[str, Optional[Tensor]]

_forward_hooks: Dict[int, Callable]

_forward_pre_hooks: Dict[int, Callable]

_is_full_backward_hook: Optional[bool]

_load_state_dict_post_hooks: Dict[int, Callable]

_load_state_dict_pre_hooks: Dict[int, Callable]

_modules: Dict[str, Optional[Module]]

_non_persistent_buffers_set: Set[str]

_parameters: Dict[str, Optional[Parameter]]

_state_dict_hooks: Dict[int, Callable]

classmethod add_config(*cfg*)

forward(*batched_inputs*, *mode=None*)

Defines the computation performed at every call.

Should be overridden 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.

---

classmethod from_config(*cfg*)

training: bool

class xmodaler.modeling.encoder.**TDConvEDEncoder**(*, *num_hidden_layers: int*, *hidden_size: int*, *kernel_sizes: list*, *padding_mode: str*, *offset_act: str*, *min_idx: int*, *max_idx: int*, *clamp_idx: bool*, *dropout: float*, *use_norm: bool*)

Bases: `Module`

__init__(*, *num_hidden_layers: int*, *hidden_size: int*, *kernel_sizes: list*, *padding_mode: str*, *offset_act: str*, *min_idx: int*, *max_idx: int*, *clamp_idx: bool*, *dropout: float*, *use_norm: bool*)

Initializes internal Module state, shared by both nn.Module and ScriptModule.

_backward_hooks: Dict[int, Callable]

_buffers: Dict[str, Optional[Tensor]]

_forward_hooks: Dict[int, Callable]

_forward_pre_hooks: Dict[int, Callable]

_is_full_backward_hook: Optional[bool]

_load_state_dict_post_hooks: Dict[int, Callable]

_load_state_dict_pre_hooks: Dict[int, Callable]

_modules: Dict[str, Optional[Module]]

_non_persistent_buffers_set: Set[str]

_parameters: Dict[str, Optional[Parameter]]

_state_dict_hooks: Dict[int, Callable]

classmethod add_config(*cfg*)

forward(*batched_inputs*, *mode=None*)

Defines the computation performed at every call.

Should be overridden 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.

classmethod from_config(*cfg*)

training: bool

3.9.5 xmodaler.modeling.layers

xmodaler.modeling.layers.create_act.**get_act_layer**(*name='none'*)

xmodaler.modeling.layers.create_act.**get_activation**(*activation_string*)

class xmodaler.modeling.layers.attention_pooler.**AttentionPooler**(*, *hidden_size: int*, *output_size: int*, *dropout: float*, *use_bn: bool*)

Bases: `Module`

__init__(*, *hidden_size: int*, *output_size: int*, *dropout: float*, *use_bn: bool*)

Initializes internal Module state, shared by both nn.Module and ScriptModule.

forward(*hidden_states*, *masks=None*, ***kwargs*)

Defines the computation performed at every call.

Should be overridden 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.

training: bool

class xmodaler.modeling.layers.base_attention.**BaseAttention**(*, *hidden_size: int*, *att_embed_size: int*, *att_embed_dropout: float*)

Bases: Module

__init__(*, *hidden_size: int*, *att_embed_size: int*, *att_embed_dropout: float*)

Initializes internal Module state, shared by both nn.Module and ScriptModule.

forward(*hidden_states*, *att_feats*, *p_att_feats*, *att_masks=None*, ***kwargs*)

Defines the computation performed at every call.

Should be overridden 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.

---

training: bool

class xmodaler.modeling.layers.bert.**BertSelfAttention**(*, *hidden_size*, *num_attention_heads*, *attention_probs_dropout_prob*)

Bases: Module

__init__(*, *hidden_size*, *num_attention_heads*, *attention_probs_dropout_prob*)

Initializes internal Module state, shared by both nn.Module and ScriptModule.

forward(*hidden_states*, *attention_mask*, *history_states=None*)

Defines the computation performed at every call.

Should be overridden 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.

---

classmethod from_config(*cfg*)

training: bool

transpose_for_scores(*x*)

class xmodaler.modeling.layers.bert.**BertSelfOutput**(*, *hidden_size: int*, *layer_norm_eps: float*, *hidden_dropout_prob: float*)

Bases: Module

__init__(*, *hidden_size: int*, *layer_norm_eps: float*, *hidden_dropout_prob: float*)

Initializes internal Module state, shared by both nn.Module and ScriptModule.

forward(*hidden_states*, *input_tensor*)

Defines the computation performed at every call.

Should be overridden 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.

classmethod from_config(*cfg*)

training: bool

class xmodaler.modeling.layers.bert.**BertAttention**(*, *bert_self_attention*, *bert_self_output*)

Bases: `Module`

__init__(*, *bert_self_attention*, *bert_self_output*)

Initializes internal Module state, shared by both nn.Module and ScriptModule.

forward(*input_tensor*, *attention_mask*, *history_states=None*)

Defines the computation performed at every call.

Should be overridden 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.

classmethod from_config(*cfg*)

training: bool

class xmodaler.modeling.layers.bert.**BertIntermediate**(*, *hidden_size: int*, *hidden_act: str*, *intermediate_size: int*, *intermediate_drop: float*)

Bases: `Module`

__init__(*, *hidden_size: int*, *hidden_act: str*, *intermediate_size: int*, *intermediate_drop: float*)

Initializes internal Module state, shared by both nn.Module and ScriptModule.

forward(*hidden_states*)

Defines the computation performed at every call.

Should be overridden 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.

classmethod from_config(*cfg*)

training: bool

class xmodaler.modeling.layers.bert.**BertOutput**(*, *hidden_size: int*, *intermediate_size: int*, *layer_norm_eps: float*, *ffn_dropout_prob: float*)

Bases: `Module`

__init__(*, *hidden_size: int*, *intermediate_size: int*, *layer_norm_eps: float*, *ffn_dropout_prob: float*)

Initializes internal Module state, shared by both nn.Module and ScriptModule.

forward(*hidden_states*, *input_tensor*)

Defines the computation performed at every call.

Should be overridden 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.

classmethod from_config(*cfg*)

training: bool

class xmodaler.modeling.layers.bert.**BertXAttention**(*, *hidden_size*, *num_attention_heads*, *attention_probs_dropout_prob*)

Bases: `Module`

__init__(*, *hidden_size*, *num_attention_heads*, *attention_probs_dropout_prob*)

Initializes internal Module state, shared by both nn.Module and ScriptModule.

forward(*query*, *key*, *value*, *attention_mask*)

Defines the computation performed at every call.

Should be overridden 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.

classmethod from_config(*cfg*)

training: bool

transpose_for_scores(*x*)

class xmodaler.modeling.layers.bert.**BertCrossAttention**(*, *bert_cross_attention*, *bert_self_output*)

Bases: `Module`

__init__(*, *bert_cross_attention*, *bert_self_output*)

Initializes internal Module state, shared by both nn.Module and ScriptModule.

forward(*query*, *key*, *value*, *attention_mask*, *q_attention_mask*)

Defines the computation performed at every call.

Should be overridden 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.

classmethod from_config(*cfg*)

training: bool

class xmodaler.modeling.layers.bert.**BertLayer**(*, *bert_attention*, *bert_intermediate*, *bert_output*)

Bases: Module

__init__(*, *bert_attention*, *bert_intermediate*, *bert_output*)

Initializes internal Module state, shared by both nn.Module and ScriptModule.

forward(*hidden_states*, *attention_mask*, *history_states=None*)

Defines the computation performed at every call.

Should be overridden 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.

---

classmethod from_config(*cfg*)

training: bool

class xmodaler.modeling.layers.bert.**BertUnderstandingLayer**(*, *bert_attention*, *v_bert_intermediate*, *v_bert_output*, *t_bert_intermediate*, *t_bert_output*)

Bases: Module

__init__(*, *bert_attention*, *v_bert_intermediate*, *v_bert_output*, *t_bert_intermediate*, *t_bert_output*)

Initializes internal Module state, shared by both nn.Module and ScriptModule.

forward(*input_tensor1*, *attention_mask1*, *input_tensor2*, *attention_mask2*)

Defines the computation performed at every call.

Should be overridden 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.

---

classmethod from_config(*cfg*)

training: bool

class xmodaler.modeling.layers.bert.**BertGenerationLayer**(*, *bert_attention*, *bert_cross_attention*, *bert_intermediate*, *bert_output*)

Bases: Module

__init__(*, *bert_attention*, *bert_cross_attention*, *bert_intermediate*, *bert_output*)

Initializes internal Module state, shared by both nn.Module and ScriptModule.

forward(*lang_feats*, *v_feats*, *lang_attention_mask=None*, *v_attention_mask=None*, *t_history_states=None*)

Defines the computation performed at every call.

Should be overridden 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.

classmethod from_config(*cfg*)

training: bool

class xmodaler.modeling.layers.bert.**BertPooler**(*, *hidden_size: int*)

Bases: `Module`

__init__(*, *hidden_size: int*)

Initializes internal Module state, shared by both nn.Module and ScriptModule.

forward(*hidden_states*)

Defines the computation performed at every call.

Should be overridden 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.

classmethod from_config(*cfg*)

training: bool

class xmodaler.modeling.layers.bert.**BertPredictionHeadTransform**(*, *hidden_size: int*, *hidden_act: str*, *layer_norm_eps: float*)

Bases: `Module`

__init__(*, *hidden_size: int*, *hidden_act: str*, *layer_norm_eps: float*)

Initializes internal Module state, shared by both nn.Module and ScriptModule.

forward(*hidden_states*)

Defines the computation performed at every call.

Should be overridden 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.

classmethod from_config(*cfg*)

training: bool

class xmodaler.modeling.layers.multihead_attention.**MultiHeadAttentionMemory**(*, *d_model: int*, *d_k: int*, *d_v: int*, *num_head: int*, *dropout: float*, *num_memory: int*)

Bases: `Module`

Multi-head attention layer with Dropout and Layer Normalization.

__init__(*, *d_model: int*, *d_k: int*, *d_v: int*, *num_head: int*, *dropout: float*, *num_memory: int*)

Initializes internal Module state, shared by both nn.Module and ScriptModule.

forward(*queries*, *keys*, *values*, *attention_mask=None*, *attention_weights=None*)

Defines the computation performed at every call.

Should be overridden 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.

training: bool

class xmodaler.modeling.layers.multihead_attention.**MultiHeadAttention**(*, *d_model: int*, *d_k: int*, *d_v: int*, *num_head: int*, *dropout: float*)

Bases: `Module`

Multi-head attention layer with Dropout and Layer Normalization.

__init__(*, *d_model: int*, *d_k: int*, *d_v: int*, *num_head: int*, *dropout: float*)

Initializes internal Module state, shared by both nn.Module and ScriptModule.

forward(*queries*, *keys*, *values*, *attention_mask=None*, *attention_weights=None*, *history_states=None*)

Defines the computation performed at every call.

Should be overridden 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.

training: bool

class xmodaler.modeling.layers.positionwise_feedforward.**PositionWiseFeedForward**(*, *d_model: int*, *d_ff: int*, *dropout: float*)

Bases: `Module`

Position-wise feed forward layer

__init__(*, *d_model: int*, *d_ff: int*, *dropout: float*)

Initializes internal Module state, shared by both nn.Module and ScriptModule.

forward(*inputs*)

Defines the computation performed at every call.

Should be overridden 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.

training: bool

class xmodaler.modeling.layers.lowrank_bilinear_layers.**LowRankBilinearAttention**(*, *embed_dim: int*, *att_heads: int*, *att_mid_dim: list*, *att_mid_drop: float*, *dropout: float*, *layer_num: int*, *act_type: str*, *elu_alpha: float*)

Bases: `Module`

__init__(*, *embed_dim: int*, *att_heads: int*, *att_mid_dim: list*, *att_mid_drop: float*, *dropout: float*, *layer_num: int*, *act_type: str*, *elu_alpha: float*)

Initializes internal Module state, shared by both nn.Module and ScriptModule.

forward(*gv_feat*, *att_feats*, *att_mask*, *p_att_feats=None*, *precompute=False*)

Defines the computation performed at every call.

Should be overridden 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.

precompute(*key*, *value2*)

training: bool

class xmodaler.modeling.layers.lowrank_bilinear_layers.**LowRankBilinearLayer**(*, *embed_dim: int*, *att_heads: int*, *att_mid_dim: list*, *att_mid_drop: float*, *dropout: float*, *act_type: str*, *elu_alpha: float*)

Bases: `Module`

__init__(*, *embed_dim: int*, *att_heads: int*, *att_mid_dim: list*, *att_mid_drop: float*, *dropout: float*, *act_type: str*, *elu_alpha: float*)

Initializes internal Module state, shared by both nn.Module and ScriptModule.

forward(*x*, *key=None*, *mask=None*, *value1=None*, *value2=None*, *precompute=False*)

Defines the computation performed at every call.

Should be overridden 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.

---

precompute(*key*, *value2*)

training: bool

class xmodaler.modeling.layers.scattention.**SCAttention**(*mid_dims: list*, *mid_dropout: float*)

Bases: `BasicAtt`

__init__(*mid_dims: list*, *mid_dropout: float*)

Initializes internal Module state, shared by both nn.Module and ScriptModule.

forward(*att_map*, *att_mask*, *value1*, *value2*)

Defines the computation performed at every call.

Should be overridden 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.

---

training: bool

class xmodaler.modeling.layers.tdconved_layers.**TemporalDeformableLayer**(*in_channels: int*, *out_channels: int*, *kernel_size: int*, *stride: int*, *padding_mode: str*, *offset_act: str*, *min_idx: int*, *max_idx: int*, *clamp_idx: bool*, *dropout: float*, *use_norm: bool*)

Bases: `Module`

__init__(*in_channels: int*, *out_channels: int*, *kernel_size: int*, *stride: int*, *padding_mode: str*, *offset_act: str*, *min_idx: int*, *max_idx: int*, *clamp_idx: bool*, *dropout: float*, *use_norm: bool*)

Initializes internal Module state, shared by both nn.Module and ScriptModule.

forward(*inputs*)

Defines the computation performed at every call.

Should be overridden 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.

training: bool

class xmodaler.modeling.layers.tdconved_layers.**TemporalDeformableBlock**(*in_channels: int*, *out_channels: int*, *kernel_size: int*, *stride: int*, *padding_mode: str*, *offset_act: str*, *min_idx: int*, *max_idx: int*, *clamp_idx: bool*, *use_norm: bool*)

Bases: `Module`

__init__(*in_channels: int*, *out_channels: int*, *kernel_size: int*, *stride: int*, *padding_mode: str*, *offset_act: str*, *min_idx: int*, *max_idx: int*, *clamp_idx: bool*, *use_norm: bool*)

Initializes internal Module state, shared by both nn.Module and ScriptModule.

forward(*inputs*)

Defines the computation performed at every call.

Should be overridden 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.

training: bool

class xmodaler.modeling.layers.tdconved_layers.**ShiftedConvLayer**(*in_channels: int*, *out_channels: int*, *kernel_size: list*, *stride: int*, *padding_mode: str*, *dropout: float*, *use_norm: bool*)

Bases: `Module`

__init__(*in_channels: int*, *out_channels: int*, *kernel_size: list*, *stride: int*, *padding_mode: str*, *dropout: float*, *use_norm: bool*)

Initializes internal Module state, shared by both nn.Module and ScriptModule.

forward(*inputs*)

Defines the computation performed at every call.

Should be overridden 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.

training: bool

class xmodaler.modeling.layers.tdconved_layers.**SoftAttention**(*, *hidden_size: int*, *att_embed_size: int*, *att_embed_dropout: float*, *use_norm: bool*)

Bases: Module

__init__(*, *hidden_size: int*, *att_embed_size: int*, *att_embed_dropout: float*, *use_norm: bool*)

Initializes internal Module state, shared by both nn.Module and ScriptModule.

forward(*hidden_states*, *att_feats*, *p_att_feats*, *att_masks=None*, ***kwargs*)

Defines the computation performed at every call.

Should be overridden 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.

training: bool

3.9.6 xmodaler.modeling.meta_arch

xmodaler.modeling.meta_arch.**build_model**(*cfg*)

Build the whole model architecture, defined by cfg.MODEL.META_ARCHITECTURE. Note that it does not load any weights from cfg.

xmodaler.modeling.meta_arch.**add_config**(*cfg*, *tmp_cfg*)

class xmodaler.modeling.meta_arch.base_enc_dec.**BaseEncoderDecoder**(*, *vocab_size*, *max_seq_len*, *token_embed*, *visual_embed*, *encoder*, *decoder*, *predictor*, *greedy_decoder*, *beam_searcher*)

Bases: Module

__init__(*, *vocab_size*, *max_seq_len*, *token_embed*, *visual_embed*, *encoder*, *decoder*, *predictor*, *greedy_decoder*, *beam_searcher*)

Initializes internal Module state, shared by both nn.Module and ScriptModule.

classmethod add_config(*cfg*, *tmp_cfg*)

bind_or_init_weights()

decode_beam_search(*batched_inputs*, *output_sents=False*)

forward(*batched_inputs*, *use_beam_search=None*, *output_sents=False*)

Defines the computation performed at every call.

Should be overridden 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.

classmethod from_config(*cfg*)

abstract get_extended_attention_mask(*batched_inputs*)

greedy_decode(*batched_inputs*, *output_sents=False*)

preprocess_batch(*batched_inputs*)

training: bool

class xmodaler.modeling.meta_arch.**RnnAttEncoderDecoder**(*, *vocab_size*, *max_seq_len*, *token_embed*, *visual_embed*, *encoder*, *decoder*, *predictor*, *greedy_decoder*, *beam_searcher*)

Bases: BaseEncoderDecoder

get_extended_attention_mask(*batched_inputs*)

training: bool

class xmodaler.modeling.meta_arch.**TransformerEncoderDecoder**(*, *vocab_size*, *max_seq_len*, *token_embed*, *visual_embed*, *encoder*, *decoder*, *predictor*, *greedy_decoder*, *beam_searcher*, *v_predictor*)

Bases: BaseEncoderDecoder

__init__(*, *vocab_size*, *max_seq_len*, *token_embed*, *visual_embed*, *encoder*, *decoder*, *predictor*, *greedy_decoder*, *beam_searcher*, *v_predictor*)

Initializes internal Module state, shared by both nn.Module and ScriptModule.

classmethod from_config(*cfg*)

get_extended_attention_mask(*batched_inputs*)

training: bool

class xmodaler.modeling.meta_arch.**TDENBiTransformer**(*, *vocab_size*, *max_seq_len*, *token_embed*, *visual_embed*, *encoder*, *decoder*, *predictor*, *greedy_decoder*, *beam_searcher*, *v_predictor*)

Bases: TransformerEncoderDecoder

__init__(*, *vocab_size*, *max_seq_len*, *token_embed*, *visual_embed*, *encoder*, *decoder*, *predictor*, *greedy_decoder*, *beam_searcher*, *v_predictor*)

Initializes internal Module state, shared by both nn.Module and ScriptModule.

classmethod from_config(*cfg*)

get_extended_attention_mask(*batched_inputs*)

training: bool

class xmodaler.modeling.meta_arch.**TDENPretrain**(*, *vocab_size*, *max_seq_len*, *token_embed*, *visual_embed*, *encoder*, *decoder*, *predictor*, *greedy_decoder*, *beam_searcher*, *v_predictor*, *similarity_predictor*)

Bases: TransformerEncoderDecoder

__init__(*, *vocab_size*, *max_seq_len*, *token_embed*, *visual_embed*, *encoder*, *decoder*, *predictor*, *greedy_decoder*, *beam_searcher*, *v_predictor*, *similarity_predictor*)

Initializes internal Module state, shared by both nn.Module and ScriptModule.

classmethod add_config(*cfg*, *tmp_cfg*)

classmethod from_config(*cfg*)

training: bool

class xmodaler.modeling.meta_arch.**TDENCaptioner**(*, *vocab_size*, *max_seq_len*, *token_embed*, *visual_embed*, *encoder*, *decoder*, *predictor*, *greedy_decoder*, *beam_searcher*, *v_predictor*)

Bases: *TransformerEncoderDecoder*

__init__(*, *vocab_size*, *max_seq_len*, *token_embed*, *visual_embed*, *encoder*, *decoder*, *predictor*, *greedy_decoder*, *beam_searcher*, *v_predictor*)

Initializes internal Module state, shared by both nn.Module and ScriptModule.

classmethod from_config(*cfg*)

training: bool

class xmodaler.modeling.meta_arch.**UniterPretrain**(*, *vocab_size*, *max_seq_len*, *token_embed*, *visual_embed*, *encoder*, *decoder*, *predictor*, *greedy_decoder*, *beam_searcher*, *v_predictor*, *v_regressor*, *itm_predictor*, *tasks*, *mix_ratio*)

Bases: *TransformerEncoderDecoder*

__init__(*, *vocab_size*, *max_seq_len*, *token_embed*, *visual_embed*, *encoder*, *decoder*, *predictor*, *greedy_decoder*, *beam_searcher*, *v_predictor*, *v_regressor*, *itm_predictor*, *tasks*, *mix_ratio*)

Initializes internal Module state, shared by both nn.Module and ScriptModule.

classmethod add_config(*cfg*, *tmp_cfg*)

classmethod from_config(*cfg*)

preprocess_inputs(*inputs*, *task_name*)

training: bool

class xmodaler.modeling.meta_arch.**UniterForMMUnderstanding**(*, *vocab_size*, *max_seq_len*, *token_embed*, *visual_embed*, *encoder*, *decoder*, *predictor*, *greedy_decoder*, *beam_searcher*, *v_predictor*, *itm_predictor*)

Bases: *TransformerEncoderDecoder*

__init__(*, *vocab_size*, *max_seq_len*, *token_embed*, *visual_embed*, *encoder*, *decoder*, *predictor*, *greedy_decoder*, *beam_searcher*, *v_predictor*, *itm_predictor*)

Initializes internal Module state, shared by both nn.Module and ScriptModule.

bind_or_init_weights()

classmethod from_config(*cfg*)

get_extended_attention_mask(*batched_inputs*)

training: bool

3.9.7 xmodaler.modeling.predictor

xmodaler.modeling.predictor.**build_predictor**(*cfg*)

xmodaler.modeling.predictor.**build_v_predictor**(*cfg*)

xmodaler.modeling.predictor.**add_predictor_config**(*cfg*, *tmp_cfg*)

xmodaler.modeling.predictor.**build_predictor_with_name**(*cfg*, *name*)

class xmodaler.modeling.predictor.**BasePredictor**(*, *hidden_size: int*, *vocab_size: int*, *dropout: float*)

Bases: Module

__init__(*, *hidden_size: int*, *vocab_size: int*, *dropout: float*)

Initializes internal Module state, shared by both nn.Module and ScriptModule.

classmethod add_config(*cfg*)

forward(*batched_inputs*)

Defines the computation performed at every call.

Should be overridden 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.

classmethod from_config(*cfg*)

training: bool

class xmodaler.modeling.predictor.**BertPredictionHead**(*, *hidden_size*, *vocab_size*, *transform*)

Bases: Module

__init__(*, *hidden_size*, *vocab_size*, *transform*)

Initializes internal Module state, shared by both nn.Module and ScriptModule.

classmethod add_config(*cfg*)

forward(*batched_inputs*)

Defines the computation performed at every call.

Should be overridden 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.

classmethod from_config(*cfg*)

training: bool

class xmodaler.modeling.predictor.**BertVisualPredictionHead**(*, *hidden_size*, *v_target_size*, *transform*)

Bases: Module

__init__(*, *hidden_size*, *v_target_size*, *transform*)

Initializes internal Module state, shared by both nn.Module and ScriptModule.

classmethod add_config(*cfg*)

forward(*batched_inputs*)

Defines the computation performed at every call.

Should be overridden 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.

classmethod from_config(*cfg*)

training: bool

class xmodaler.modeling.predictor.**BertVisualFeatureRegressionHead**(*, *hidden_size*, *v_feat_dim*, *transform*)

Bases: `Module`

__init__(*, *hidden_size*, *v_feat_dim*, *transform*)

Initializes internal Module state, shared by both nn.Module and ScriptModule.

classmethod add_config(*cfg*)

forward(*batched_inputs*)

Defines the computation performed at every call.

Should be overridden 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.

classmethod from_config(*cfg*)

training: bool

class xmodaler.modeling.predictor.**BertIsMatchedPredictor**(*, *hidden_size*, *pooler*)

Bases: `Module`

__init__(*, *hidden_size*, *pooler*)

Initializes internal Module state, shared by both nn.Module and ScriptModule.

classmethod add_config(*cfg*)

forward(*batched_inputs*)

Defines the computation performed at every call.

Should be overridden 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.

classmethod from_config(*cfg*)

training: bool

class xmodaler.modeling.predictor.**MultiModalPredictor**(*, *labels_num: int*, *pooler_input_size: int*, *pooler_output_size: int*, *pooler_bn: bool*, *pooler_dropout: float*, *num_understanding_layers: int*, *num_generation_layers: int*)

Bases: `Module`

__init__(*, *labels_num: int*, *pooler_input_size: int*, *pooler_output_size: int*, *pooler_bn: bool*, *pooler_dropout: float*, *num_understanding_layers: int*, *num_generation_layers: int*)

Initializes internal Module state, shared by both nn.Module and ScriptModule.

classmethod add_config(*cfg*)

forward(*batched_inputs*)

Defines the computation performed at every call.

Should be overridden 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.

classmethod from_config(*cfg*)

training: bool

class xmodaler.modeling.predictor.**SingleStreamMultiModalPredictor**(*, *hidden_size: int*, *labels_num: int*, *pooler*)

Bases: `Module`

__init__(*, *hidden_size: int*, *labels_num: int*, *pooler*)

Initializes internal Module state, shared by both nn.Module and ScriptModule.

classmethod add_config(*cfg*)

forward(*batched_inputs*)

Defines the computation performed at every call.

Should be overridden 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.

classmethod from_config(*cfg*)

test_forward(*u_logits*)

training: bool

class xmodaler.modeling.predictor.**MultiModalSimilarity**(*, *pooler_input_size: int*, *pooler_output_size: int*, *pooler_bn: bool*, *pooler_dropout: float*, *num_hidden_layers: int*, *v_num_hidden_layers: int*)

Bases: Module

__init__(*, *pooler_input_size: int*, *pooler_output_size: int*, *pooler_bn: bool*, *pooler_dropout: float*, *num_hidden_layers: int*, *v_num_hidden_layers: int*)

Initializes internal Module state, shared by both nn.Module and ScriptModule.

classmethod add_config(*cfg*)

forward(*batched_inputs*)

Defines the computation performed at every call.

Should be overridden 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.

classmethod from_config(*cfg*)

training: bool

3.10 xmodaler.optim

xmodaler.optim.**build_optimizer**(*cfg:* CfgNode, *model: Module*) → Optimizer

Build an optimizer from config.

class xmodaler.optim.**Adagrad**(*, *params*, *lr=0.01*, *lr_decay=0*, *weight_decay=0*, *initial_accumulator_value=0*, *eps=1e-10*)

Bases: Adagrad

__init__(*, *params*, *lr=0.01*, *lr_decay=0*, *weight_decay=0*, *initial_accumulator_value=0*, *eps=1e-10*)

classmethod from_config(*cfg*, *params*)

class xmodaler.optim.**Adam**(*, *params*, *lr=0.001*, *betas=(0.9, 0.999)*, *eps=1e-08*, *weight_decay=0*, *amsgrad=False*)

Bases: Adam

__init__(*, *params*, *lr=0.001*, *betas=(0.9, 0.999)*, *eps=1e-08*, *weight_decay=0*, *amsgrad=False*)

classmethod from_config(*cfg*, *params*)

class xmodaler.optim.**AdamW**(*, *params*, *lr=0.001*, *betas=(0.9, 0.999)*, *eps=1e-08*, *weight_decay=0.01*, *amsgrad=False*)

Bases: AdamW

__init__(*, *params*, *lr=0.001*, *betas=(0.9, 0.999)*, *eps=1e-08*, *weight_decay=0.01*, *amsgrad=False*)

classmethod from_config(*cfg*, *params*)

class xmodaler.optim.**Adamax**(*, *params*, *lr=0.002*, *betas=(0.9, 0.999)*, *eps=1e-08*, *weight_decay=0*)

Bases: Adamax

__init__(*, *params*, *lr=0.002*, *betas=(0.9, 0.999)*, *eps=1e-08*, *weight_decay=0*)

classmethod from_config(*cfg*, *params*)

class xmodaler.optim.**RAdam**(*, *params*, *lr=0.001*, *betas=(0.9, 0.999)*, *eps=1e-08*, *weight_decay=0*)

Bases: Optimizer

__init__(*, *params*, *lr=0.001*, *betas=(0.9, 0.999)*, *eps=1e-08*, *weight_decay=0*)

classmethod from_config(*cfg*, *params*)

step(*closure=None*)

Performs a single optimization step (parameter update).

Parameters

closure (*Callable*) – A closure that reevaluates the model and returns the loss. Optional for most optimizers.

Note: Unless otherwise specified, this function should not modify the .grad field of the parameters.

class xmodaler.optim.**RMSprop**(*, *params*, *lr=0.01*, *alpha=0.99*, *eps=1e-08*, *weight_decay=0*, *momentum=0*, *centered=False*)

Bases: RMSprop

__init__(*, *params*, *lr=0.01*, *alpha=0.99*, *eps=1e-08*, *weight_decay=0*, *momentum=0*, *centered=False*)

classmethod from_config(*cfg*, *params*)

class xmodaler.optim.**SGD**(*, *params*, *lr=0.1*, *momentum=0*, *dampening=0*, *weight_decay=0*, *nesterov=False*)

Bases: SGD

__init__(*, *params*, *lr=0.1*, *momentum=0*, *dampening=0*, *weight_decay=0*, *nesterov=False*)

classmethod from_config(*cfg*, *params*)

class xmodaler.optim.**BertAdam**(*, *params*, *lr=0.001*, *b1=0.9*, *b2=0.999*, *eps=1e-06*, *weight_decay=0.01*, *max_grad_norm=1.0*)

Bases: Optimizer

__init__(*, *params*, *lr=0.001*, *b1=0.9*, *b2=0.999*, *eps=1e-06*, *weight_decay=0.01*, *max_grad_norm=1.0*)

classmethod from_config(*cfg*, *params*)

step(*closure=None*)

Performs a single optimization step. :param closure: A closure that reevaluates the model

and returns the loss.

3.11 xmodaler.scorer

`xmodaler.scorer.build_scorer(cfg)`

class `xmodaler.scorer.BaseScorer(*, types, scorers, weights, gt_path, eos_id)`

Bases: `object`

`__init__(*, types, scorers, weights, gt_path, eos_id)`

classmethod `from_config(cfg)`

`get_sents(sent)`

class `xmodaler.scorer.Cider(*, n: int, sigma: float, cider_cached: str)`

Bases: `object`

Main Class to compute the CIDEr metric

`__init__(*, n: int, sigma: float, cider_cached: str)`

`compute_score(gts, res)`

Main function to compute CIDEr score :param hypo_for_image (dict) : dictionary with key <image> and value <tokenized hypothesis / candidate sentence>

ref_for_image (dict) : dictionary with key <image> and value <tokenized reference sentence>

Returns
cider (float) : computed CIDEr score for the corpus

classmethod `from_config(cfg)`

`method()`

class `xmodaler.scorer.BertTokenizedScorer(*, types, scorers, weights, gt_path, bert_tokenizer)`

Bases: `object`

`__init__(*, types, scorers, weights, gt_path, bert_tokenizer)`

classmethod `from_config(cfg)`

`get_sents(sent)`

3.12 xmodaler.tokenization

class `xmodaler.tokenization.BertTokenizer(vocab_file, do_lower_case=True, do_basic_tokenize=True, never_split=None, unk_token='[UNK]', sep_token='[SEP]', pad_token='[PAD]', cls_token='[CLS]', mask_token='[MASK]', tokenize_chinese_chars=True, **kwargs)`

Bases: `PreTrainedTokenizer`

Constructs a BertTokenizer. `BertTokenizer` runs end-to-end tokenization: punctuation splitting + wordpiece

Parameters

- **vocab_file** – Path to a one-wordpiece-per-line vocabulary file

- **do_lower_case** – Whether to lower case the input. Only has an effect when do_wordpiece_only=False
- **do_basic_tokenize** – Whether to do basic tokenization before wordpiece.
- **max_len** – An artificial maximum length to truncate tokenized sequences to; Effective maximum length is always the minimum of this value (if specified) and the underlying BERT model's sequence length.
- **never_split** – List of tokens which will never be split during tokenization. Only has an effect when do_wordpiece_only=False

add_special_tokens_sentences_pair(*token_ids_0*, *token_ids_1*)

Adds special tokens to a sequence pair for sequence classification tasks. A BERT sequence pair has the following format: [CLS] A [SEP] B [SEP]

add_special_tokens_single_sentence(*token_ids*)

Adds special tokens to the a sequence for sequence classification tasks. A BERT sequence has the following format: [CLS] X [SEP]

convert_tokens_to_string(*tokens*)

Converts a sequence of tokens (string) in a single string.

```
max_model_input_sizes = {'bert-base-cased': 512, 'bert-base-cased-finetuned-mrpc':
512, 'bert-base-chinese': 512, 'bert-base-german-cased': 512,
'bert-base-multilingual-cased': 512, 'bert-base-multilingual-uncased': 512,
'bert-base-uncased': 512, 'bert-large-cased': 512,
'bert-large-cased-whole-word-masking': 512,
'bert-large-cased-whole-word-masking-finetuned-squad': 512, 'bert-large-uncased':
512, 'bert-large-uncased-whole-word-masking': 512,
'bert-large-uncased-whole-word-masking-finetuned-squad': 512}
```

```
pretrained_init_configuration = {'bert-base-cased': {'do_lower_case': False},
'bert-base-cased-finetuned-mrpc': {'do_lower_case': False}, 'bert-base-chinese':
{'do_lower_case': False}, 'bert-base-german-cased': {'do_lower_case': False},
'bert-base-multilingual-cased': {'do_lower_case': False},
'bert-base-multilingual-uncased': {'do_lower_case': True}, 'bert-base-uncased':
{'do_lower_case': True}, 'bert-large-cased': {'do_lower_case': False},
'bert-large-cased-whole-word-masking': {'do_lower_case': False},
'bert-large-cased-whole-word-masking-finetuned-squad': {'do_lower_case': False},
'bert-large-uncased': {'do_lower_case': True},
'bert-large-uncased-whole-word-masking': {'do_lower_case': True},
'bert-large-uncased-whole-word-masking-finetuned-squad': {'do_lower_case': True}}
```

```
pretrained_vocab_files_map = {'vocab_file': {'bert-base-cased':
'https://s3.amazonaws.com/models.huggingface.co/bert/bert-base-cased-vocab.txt',
'bert-base-cased-finetuned-mrpc': 'https://s3.amazonaws.com/models.huggingface.co/
bert/bert-base-cased-finetuned-mrpc-vocab.txt', 'bert-base-chinese':
'https://s3.amazonaws.com/models.huggingface.co/bert/bert-base-chinese-vocab.txt',
'bert-base-german-cased': 'https://int-deepset-models-bert.s3.eu-central-1.
amazonaws.com/pytorch/bert-base-german-cased-vocab.txt',
'bert-base-multilingual-cased': 'https://s3.amazonaws.com/models.huggingface.co/
bert/bert-base-multilingual-cased-vocab.txt', 'bert-base-multilingual-uncased':
'https://s3.amazonaws.com/models.huggingface.co/bert/
bert-base-multilingual-uncased-vocab.txt', 'bert-base-uncased':
'../pretrain/BERT/bert-base-uncased-vocab.txt', 'bert-large-cased':
'https://s3.amazonaws.com/models.huggingface.co/bert/bert-large-cased-vocab.txt',
'bert-large-cased-whole-word-masking': 'https://s3.amazonaws.com/models.
huggingface.co/bert/bert-large-cased-whole-word-masking-vocab.txt',
'bert-large-cased-whole-word-masking-finetuned-squad':
'https://s3.amazonaws.com/models.huggingface.co/bert/
bert-large-cased-whole-word-masking-finetuned-squad-vocab.txt',
'bert-large-uncased':
'https://s3.amazonaws.com/models.huggingface.co/bert/bert-large-uncased-vocab.txt',
'bert-large-uncased-whole-word-masking': 'https://s3.amazonaws.com/models.
huggingface.co/bert/bert-large-uncased-whole-word-masking-vocab.txt',
'bert-large-uncased-whole-word-masking-finetuned-squad':
'https://s3.amazonaws.com/models.huggingface.co/bert/
bert-large-uncased-whole-word-masking-finetuned-squad-vocab.txt'}}
```

save_vocabulary(*vocab_path*)

Save the tokenizer vocabulary to a directory or file.

vocab_files_names = {'vocab_file': 'vocab.txt'}

property vocab_size

Size of the base vocabulary (without the added tokens)

3.13 xmodaler.utils

xmodaler.utils.collect_env.**collect_env_info**()

xmodaler.utils.colormap.**colormap**(*rgb=False*, *maximum=255*)

Parameters

- **rgb** (*bool*) – whether to return RGB colors or BGR colors.
- **maximum** (*int*) – either 255 or 1

Returns

a float32 array of Nx3 colors, in range [0, 255] or [0, 1]

Return type

ndarray

xmodaler.utils.colormap.**random_color**(*rgb=False*, *maximum=255*)

Parameters

- **rgb** (*bool*) – whether to return RGB colors or BGR colors.
- **maximum** (*int*) – either 255 or 1

Returns
a vector of 3 numbers

Return type
ndarray

`xmodaler.utils.comm.all_gather(data, group=None)`

Run all_gather on arbitrary picklable data (not necessarily tensors).

Parameters

- **data** – any picklable object
- **group** – a torch process group. By default, will use a group which contains all ranks on gloo backend.

Returns
list of data gathered from each rank

Return type
list[data]

`xmodaler.utils.comm.gather(data, dst=0, group=None)`

Run gather on arbitrary picklable data (not necessarily tensors).

Parameters

- **data** – any picklable object
- **dst** (*int*) – destination rank
- **group** – a torch process group. By default, will use a group which contains all ranks on gloo backend.

Returns

on dst, a list of data gathered from each rank. Otherwise,
an empty list.

Return type
list[data]

`xmodaler.utils.distributed.all_gather_list(data)`

Gathers arbitrary data from all nodes into a list.

. autofunction:: xmodaler.utils.comm.any_broadcast

`xmodaler.utils.comm.get_local_rank() → int`

Returns
The rank of the current process within the local (per-machine) process group.

`xmodaler.utils.comm.get_local_size() → int`

Returns
The size of the per-machine process group, i.e. the number of processes per machine.

`xmodaler.utils.comm.get_rank() → int`

`xmodaler.utils.comm.get_world_size() → int`

`xmodaler.utils.comm.is_main_process()` → bool

`xmodaler.utils.comm.reduce_dict(input_dict, average=True)`

Reduce the values in the dictionary from all processes so that process with rank 0 has the reduced results.

Parameters

- **input_dict** (*dict*) – inputs to be reduced. All the values must be scalar CUDA Tensor.
- **average** (*bool*) – whether to do average or sum

Returns
a dict with the same keys as input_dict, after reduction.

`xmodaler.utils.comm.shared_random_seed()`

Returns

a random number that is the same across all workers.
If workers need a shared RNG, they can use this shared seed to create one.

Return type
int

All workers must call this function, otherwise it will deadlock.

`xmodaler.utils.comm.synchronize()`

Helper function to synchronize (barrier) among all processes when using distributed training

`xmodaler.utils.comm.unwrap_model(model)`

`xmodaler.utils.env.seed_all_rng(seed=None)`

Set the random seed for the RNG in torch, numpy and python.

Parameters
seed (*int*) – if None, will use a strong random seed.

`xmodaler.utils.env.setup_environment()`

Perform environment setup work. The default setup is a no-op, but this function allows the user to specify a Python source file or a module in the $DETECTRON2_ENV_MODULE environment variable, that performs custom setup work that may be necessary to their computing environment.

`xmodaler.utils.env.setup_custom_environment(custom_module)`

Load custom environment setup by importing a Python source file or a module, and run the setup function.

`xmodaler.utils.events.get_event_storage()`

Returns
The *EventStorage* object that's currently being used. Throws an error if no *EventStorage* is currently enabled.

class `xmodaler.utils.events.JSONWriter(json_file, window_size=20)`

Bases: `EventWriter`

Write scalars to a json file.

It saves scalars as one json per line (instead of a big json) for easy parsing.

Examples parsing such a json file:

```
$ cat metrics.json | jq -s '.[0:2]'
[
  {
    "data_time": 0.008433341979980469,
    "iteration": 19,
    "loss": 1.9228371381759644,
    "loss_box_reg": 0.050025828182697296,
    "loss_classifier": 0.5316952466964722,
    "loss_mask": 0.7236229181289673,
    "loss_rpn_box": 0.0856662318110466,
    "loss_rpn_cls": 0.48198649287223816,
    "lr": 0.007173333333333333,
    "time": 0.25401854515075684
  },
  {
    "data_time": 0.007216215133666992,
    "iteration": 39,
    "loss": 1.282649278640747,
    "loss_box_reg": 0.06222952902317047,
    "loss_classifier": 0.30682939291000366,
    "loss_mask": 0.6970193982124329,
    "loss_rpn_box": 0.038663312792778015,
    "loss_rpn_cls": 0.1471673548221588,
    "lr": 0.007706666666666667,
    "time": 0.2490077018737793
  }
]

$ cat metrics.json | jq '.loss_mask'
0.7126231789588928
0.689423680305481
0.6776131987571716
...
```

__init__(*json_file*, *window_size=20*)

Parameters

- **json_file** (*str*) – path to the json file. New data will be appended if the file exists.
- **window_size** (*int*) – the window size of median smoothing for the scalars whose *smoothing_hint* are True.

close()

write()

class xmodaler.utils.events.**TensorboardXWriter**(*log_dir: str*, *window_size: int = 20*, ***kwargs*)

Bases: EventWriter

Write all scalars to a tensorboard file.

__init__(*log_dir: str*, *window_size: int = 20*, ***kwargs*)

Parameters

- **log_dir** (*str*) – the directory to save the output events

- **window_size** (*int*) – the scalars will be median-smoothed by this window size
- **kwargs** – other arguments passed to *torch.utils.tensorboard.SummaryWriter(...)*

close()

write()

class xmodaler.utils.events.**CommonMetricPrinter**(*max_iter: Optional[int] = None*)

Bases: EventWriter

Print **common** metrics to the terminal, including iteration time, ETA, memory, all losses, and the learning rate. It also applies smoothing using a window of 20 elements.

It's meant to print common metrics in common ways. To print something in more customized ways, please implement a similar printer by yourself.

__init__(*max_iter: Optional[int] = None*)

Parameters
max_iter – the maximum number of iterations to train. Used to compute ETA. If not given, ETA will not be printed.

write()

class xmodaler.utils.events.**EventStorage**(*start_iter=0*)

Bases: object

The user-facing class that provides metric storage functionalities.

In the future we may add support for storing / logging other types of data if needed.

__init__(*start_iter=0*)

Parameters
start_iter (*int*) – the iteration number to start with

clear_histograms()

Delete all the stored histograms for visualization. This should be called after histograms are written to tensorboard.

clear_images()

Delete all the stored images for visualization. This should be called after images are written to tensorboard.

histories()

Returns
the HistoryBuffer for all scalars

Return type
dict[name -> HistoryBuffer]

history(*name*)

Returns
the scalar history for name

Return type
HistoryBuffer

property iter

Returns: int: The current iteration number. When used together with a trainer,

this is ensured to be the same as trainer.iter.

property iteration

latest()

Returns

mapping from the name of each scalar to the most
recent value and the iteration number its added.

Return type
dict[str -> (float, int)]

latest_with_smoothing_hint(*window_size=20*)

Similar to `latest()`, but the returned values are either the un-smoothed original latest value, or a median of the given window_size, depend on whether the smoothing_hint is True.

This provides a default behavior that other writers can use.

name_scope(*name*)

Yields
A context within which all the events added to this storage will be prefixed by the name scope.

put_histogram(*hist_name*, *hist_tensor*, *bins=1000*)

Create a histogram from a tensor.

Parameters

- **hist_name** (*str*) – The name of the histogram to put into tensorboard.
- **hist_tensor** (*torch.Tensor*) – A Tensor of arbitrary shape to be converted into a histogram.
- **bins** (*int*) – Number of histogram bins.

put_image(*img_name*, *img_tensor*)

Add an *img_tensor* associated with *img_name*, to be shown on tensorboard.

Parameters

- **img_name** (*str*) – The name of the image to put into tensorboard.
- **img_tensor** (*torch.Tensor or numpy.array*) – An *uint8* or *float* Tensor of shape *[channel, height, width]* where *channel* is 3. The image format should be RGB. The elements in img_tensor can either have values in [0, 1] (float32) or [0, 255] (uint8). The *img_tensor* will be visualized in tensorboard.

put_scalar(*name*, *value*, *smoothing_hint=True*)

Add a scalar *value* to the *HistoryBuffer* associated with *name*.

Parameters
smoothing_hint (*bool*) – a 'hint' on whether this scalar is noisy and should be smoothed when logged. The hint will be accessible through `EventStorage.smoothing_hints()`. A writer may ignore the hint and apply custom smoothing rule.

It defaults to True because most scalars we save need to be smoothed to provide any useful signal.

put_scalars(*, *smoothing_hint=True*, ***kwargs*)

Put multiple scalars from keyword arguments.

Examples

storage.put_scalars(loss=my_loss, accuracy=my_accuracy, smoothing_hint=True)

smoothing_hints()

Returns

the user-provided hint on whether the scalar
is noisy and needs smoothing.

Return type
dict[name -> bool]

step()

User should either: (1) Call this function to increment storage.iter when needed. Or (2) Set *storage.iter* to the correct iteration number before each iteration.

The storage will then be able to associate the new data with an iteration number.

class xmodaler.utils.file_io.**PathHandler**(*async_executor: Optional[Executor] = None*)

Bases: EventLogger

PathHandler is a base class that defines common I/O functionality for a URI protocol. It routes I/O for a generic URI which may look like "protocol://*" or a canonical filepath "/foo/bar/baz".

__init__(*async_executor: Optional[Executor] = None*) → None

When registering a *PathHandler*, the user can optionally pass in a *Executor* to run the asynchronous file operations. NOTE: For regular non-async operations of *PathManager*, there is no need to pass *async_executor*.

Parameters

async_executor (optional *Executor*) – Used for async file operations. Usage: ```

path_handler = NativePathHandler(async_executor=exe)
path_manager.register_handler(path_handler)

```

xmodaler.utils.file_io.**PathManager**

This is a detectron2 project-specific PathManager. We try to stay away from global PathManager in fvcore as it introduces potential conflicts among other libraries.

**class** xmodaler.utils.logger.**Counter**(***kwds*)

Bases: dict

Dict subclass for counting hashable items. Sometimes called a bag or multiset. Elements are stored as dictionary keys and their counts are stored as dictionary values.

```
>>> c = Counter('abcdeabcdabcaba') # count elements from a string
```

```
>>> c.most_common(3)                # three most common elements
[('a', 5), ('b', 4), ('c', 3)]
>>> sorted(c)                       # list all unique elements
['a', 'b', 'c', 'd', 'e']
```

(continues on next page)
```

(continued from previous page)

```
>>> ''.join(sorted(c.elements()))   # list elements with repetitions
'aaaaabbbbcccdde'
>>> sum(c.values)                   # total of all counts
15
```

```
>>> c['a']                          # count of letter 'a'
5
>>> for elem in 'shazam':           # update counts from an iterable
...     c[elem] += 1                # by adding 1 to each element's count
>>> c['a']                          # now there are seven 'a'
7
>>> del c['b']                      # remove all 'b'
>>> c['b']                          # now there are zero 'b'
0
```

```
>>> d = Counter('simsalabim')       # make another counter
>>> c.update(d)                     # add in the second counter
>>> c['a']                          # now there are nine 'a'
9
```

```
>>> c.clear()                       # empty the counter
>>> c
Counter()
```

Note: If a count is set to zero or reduced to zero, it will remain in the counter until the entry is deleted or the counter is cleared:

```
>>> c = Counter('aaabbc')
>>> c['b'] -= 2                     # reduce the count of 'b' by two
>>> c.most_common()                 # 'b' is still in, but its count is zero
[('a', 3), ('c', 1), ('b', 0)]
```

__init__(***kwds*)

Create a new, empty Counter object. And if given, count elements from an input iterable. Or, initialize the count from another mapping of elements to their counts.

```
>>> c = Counter()                           # a new, empty counter
>>> c = Counter('gallahad')                 # a new counter from an iterable
>>> c = Counter({'a': 4, 'b': 2})           # a new counter from a mapping
>>> c = Counter(a=4, b=2)                   # a new counter from keyword args
```

copy()

Return a shallow copy.

elements()

Iterator over elements repeating each as many times as its count.

```
>>> c = Counter('ABCABC')
>>> sorted(c.elements())
['A', 'A', 'B', 'B', 'C', 'C']
```

Knuth's example for prime factors of 1836: 2**2 * 3**3 * 17**1 >>> prime_factors = Counter({2: 2, 3:

3, 17: 1}) >>> product = 1 >>> for factor in prime_factors.elements(): # loop over factors … product *= factor # and multiply them >>> product 1836

Note, if an element's count has been set to zero or is a negative number, elements() will ignore it.

classmethod fromkeys(*iterable*, *v=None*)

Create a new dictionary with keys from iterable and values set to value.

most_common(*n=None*)

List the n most common elements and their counts from the most common to the least. If n is None, then list all element counts.

```
>>> Counter('abcdeabcdabcaba').most_common(3)
[('a', 5), ('b', 4), ('c', 3)]
```

subtract(***kwds*)

Like dict.update() but subtracts counts instead of replacing them. Counts can be reduced below zero. Both the inputs and outputs are allowed to contain zero and negative counts.

Source can be an iterable, a dictionary, or another Counter instance.

```
>>> c = Counter('which')
>>> c.subtract('witch')             # subtract elements from another iterable
>>> c.subtract(Counter('watch'))    # subtract elements from another counter
>>> c['h']                          # 2 in which, minus 1 in witch, minus 1 in␣
↪watch
0
>>> c['w']                          # 1 in which, minus 1 in witch, minus 1 in␣
↪watch
-1
```

update(***kwds*)

Like dict.update() but add counts instead of replacing them.

Source can be an iterable, a dictionary, or another Counter instance.

```
>>> c = Counter('which')
>>> c.update('witch')           # add elements from another iterable
>>> d = Counter('watch')
>>> c.update(d)                 # add elements from another counter
>>> c['h']                      # four 'h' in which, witch, and watch
4
```

xmodaler.utils.logger.**colored**(*text: str*, *color: str | None = None*, *on_color: str | None = None*, *attrs: Iterable[str] | None = None*) → str

Colorize text.

Available text colors:

black, red, green, yellow, blue, magenta, cyan, white, light_grey, dark_grey, light_red, light_green, light_yellow, light_blue, light_magenta, light_cyan.

Available text highlights:

on_black, on_red, on_green, on_yellow, on_blue, on_magenta, on_cyan, on_white, on_light_grey, on_dark_grey, on_light_red, on_light_green, on_light_yellow, on_light_blue, on_light_magenta, on_light_cyan.

Available attributes:

bold, dark, underline, blink, reverse, concealed.

Example

colored('Hello, World!', 'red', 'on_black', ['bold', 'blink']) colored('Hello, World!', 'green')

`xmodaler.utils.logger.create_small_table(small_dict)`

Create a small table using the keys of small_dict as headers. This is only suitable for small dictionaries.

Parameters

small_dict (*dict*) – a result dictionary of only a few items.

Returns

the table as a string.

Return type

str

`xmodaler.utils.logger.log_every_n(lvl, msg, n=1, *, name=None)`

Log once per n times.

Parameters

- **lvl** (*int*) – the logging level
- **msg** (*str*) –
- **n** (*int*) –
- **name** (*str*) – name of the logger to use. Will use the caller's module by default.

`xmodaler.utils.logger.log_every_n_seconds(lvl, msg, n=1, *, name=None)`

Log no more than once per n seconds.

Parameters

- **lvl** (*int*) – the logging level
- **msg** (*str*) –
- **n** (*int*) –
- **name** (*str*) – name of the logger to use. Will use the caller's module by default.

`xmodaler.utils.logger.log_first_n(lvl, msg, n=1, *, name=None, key='caller')`

Log only for the first n times.

Parameters

- **lvl** (*int*) – the logging level
- **msg** (*str*) –
- **n** (*int*) –
- **name** (*str*) – name of the logger to use. Will use the caller's module by default.
- **key** (*str or tuple[str]*) – the string(s) can be one of "caller" or "message", which defines how to identify duplicated logs. For example, if called with *n=1, key="caller"*, this function will only log the first call from the same caller, regardless of the message content. If called with *n=1, key="message"*, this function will log the same content only once, even if they are called from different places. If called with *n=1, key=("caller", "message")*, this function will not log only if the same caller has logged the same message before.

`xmodaler.utils.logger.setup_logger(output=None, distributed_rank=0, *, color=True, name='xmodaler', abbrev_name=None)`

Initialize the xmodaler logger and set its verbosity level to "DEBUG".

Parameters

- **output** (*str*) – a file name or a directory to save log. If None, will not save log file. If ends with ".txt" or ".log", assumed to be a file name. Otherwise, logs will be saved to *output/log.txt*.
- **name** (*str*) – the root module name of this logger
- **abbrev_name** (*str*) – an abbreviation of the module, to avoid long names in logs. Set to "" to not log the root module in logs. By default, will abbreviate "xmodaler" to "d2" and leave other modules unchanged.

Returns
a logger

Return type
logging.Logger

`xmodaler.utils.logger.tabulate(tabular_data, headers=(), tablefmt='simple', floatfmt='g', intfmt='', numalign='default', stralign='default', missingval='', showindex='default', disable_numparse=False, colalign=None, maxcolwidths=None, rowalign=None, maxheadercolwidths=None)`

Format a fixed width table for pretty printing.

```
>>> print(tabulate([[1, 2.34], [-56, "8.999"], ["2", "10001"]]))
---  ---------
  1      2.34
-56      8.999
  2  10001
---  ---------
```

The first required argument (*tabular_data*) can be a list-of-lists (or another iterable of iterables), a list of named tuples, a dictionary of iterables, an iterable of dictionaries, an iterable of dataclasses (Python 3.7+), a two-dimensional NumPy array, NumPy record array, or a Pandas' dataframe.

3.13.1 Table headers

To print nice column headers, supply the second argument (*headers*):

- *headers* can be an explicit list of column headers
- if *headers="firstrow"*, then the first row of data is used
- if *headers="keys"*, then dictionary keys or column indices are used

Otherwise a headerless table is produced.

If the number of headers is less than the number of columns, they are supposed to be names of the last columns. This is consistent with the plain-text format of R and Pandas' dataframes.

```
>>> print(tabulate([["sex","age"],["Alice","F",24],["Bob","M",19]],
...       headers="firstrow"))
       sex      age
-----  -----  -----
```

(continues on next page)

(continued from previous page)

```
Alice  F         24
Bob    M         19
```

By default, pandas.DataFrame data have an additional column called row index. To add a similar column to all other types of data, use *showindex="always"* or *showindex=True*. To suppress row indices for all types of data, pass *showindex="never" or `showindex=False*. To add a custom row index column, pass *showindex=some_iterable*.

```
>>> print(tabulate([["F",24],["M",19]], showindex="always"))
-  -  --
0  F  24
1  M  19
-  -  --
```

3.13.2 Column alignment

tabulate tries to detect column types automatically, and aligns the values properly. By default it aligns decimal points of the numbers (or flushes integer numbers to the right), and flushes everything else to the left. Possible column alignments (*numalign*, *stralign*) are: "right", "center", "left", "decimal" (only for *numalign*), and None (to disable alignment).

3.13.3 Table formats

intfmt is a format specification used for columns which contain numeric data without a decimal point. This can also be a list or tuple of format strings, one per column.

floatfmt is a format specification used for columns which contain numeric data with a decimal point. This can also be a list or tuple of format strings, one per column.

None values are replaced with a *missingval* string (like *floatfmt*, this can also be a list of values for different columns):

```
>>> print(tabulate([["spam", 1, None],
...                 ["eggs", 42, 3.14],
...                 ["other", None, 2.7]], missingval="?"))
-----  --  ----
spam    1  ?
eggs   42  3.14
other   ?  2.7
-----  --  ----
```

Various plain-text table formats (*tablefmt*) are supported: 'plain', 'simple', 'grid', 'pipe', 'orgtbl', 'rst', 'mediawiki', 'latex', 'latex_raw', 'latex_booktabs', 'latex_longtable' and tsv. Variable `tabulate_formats`contains the list of currently supported formats.

"plain" format doesn't use any pseudographics to draw tables, it separates columns with a double space:

```
>>> print(tabulate([["spam", 41.9999], ["eggs", "451.0"]],
...                 ["strings", "numbers"], "plain"))
strings      numbers
spam         41.9999
eggs        451
```

```
>>> print(tabulate([["spam", 41.9999], ["eggs", "451.0"]], tablefmt="plain"))
spam   41.9999
eggs  451
```

"simple" format is like Pandoc simple_tables:

```
>>> print(tabulate([["spam", 41.9999], ["eggs", "451.0"]],
...                ["strings", "numbers"], "simple"))
strings      numbers
---------  ---------
spam         41.9999
eggs        451
```

```
>>> print(tabulate([["spam", 41.9999], ["eggs", "451.0"]], tablefmt="simple"))
----  --------
spam   41.9999
eggs  451
----  --------
```

"grid" is similar to tables produced by Emacs table.el package or Pandoc grid_tables:

```
>>> print(tabulate([["spam", 41.9999], ["eggs", "451.0"]],
...               ["strings", "numbers"], "grid"))
+-----------+-----------+
| strings   |   numbers |
+===========+===========+
| spam      |   41.9999 |
+-----------+-----------+
| eggs      |  451      |
+-----------+-----------+
```

```
>>> print(tabulate([["spam", 41.9999], ["eggs", "451.0"]], tablefmt="grid"))
+------+----------+
| spam |  41.9999 |
+------+----------+
| eggs | 451      |
+------+----------+
```

"simple_grid" draws a grid using single-line box-drawing characters:

```
>>> print(tabulate([["spam", 41.9999], ["eggs", "451.0"]],
...               ["strings", "numbers"], "simple_grid"))
╒═══════════╤═══════════╕
│ strings   │   numbers │
├───────────┼───────────┤
│ spam      │   41.9999 │
├───────────┼───────────┤
│ eggs      │  451      │
╘═══════════╧═══════════╛
```

"rounded_grid" draws a grid using single-line box-drawing characters with rounded corners:

```
>>> print(tabulate([["spam", 41.9999], ["eggs", "451.0"]],
...                ["strings", "numbers"], "rounded_grid"))
───────────┬───────────
│ strings   │   numbers │
├───────────┼───────────┤
│ spam      │   41.9999 │
├───────────┼───────────┤
│ eggs      │  451      │
───────────┴───────────
```

“heavy_grid” draws a grid using bold (thick) single-line box-drawing characters:

```
>>> print(tabulate([["spam", 41.9999], ["eggs", "451.0"]],
...                ["strings", "numbers"], "heavy_grid"))
┏━━━━━━━━━━━┳━━━━━━━━━━━┓
┃ strings   ┃   numbers ┃
┣━━━━━━━━━━━╋━━━━━━━━━━━┫
┃ spam      ┃   41.9999 ┃
┣━━━━━━━━━━━╋━━━━━━━━━━━┫
┃ eggs      ┃  451      ┃
┗━━━━━━━━━━━┻━━━━━━━━━━━┛
```

“mixed_grid” draws a grid using a mix of light (thin) and heavy (thick) lines box-drawing characters:

```
>>> print(tabulate([["spam", 41.9999], ["eggs", "451.0"]],
...                ["strings", "numbers"], "mixed_grid"))
┍━━━━━━━━━━━┯━━━━━━━━━━━┑
│ strings   │   numbers │
┝━━━━━━━━━━━┿━━━━━━━━━━━┥
│ spam      │   41.9999 │
├───────────┼───────────┤
│ eggs      │  451      │
┕━━━━━━━━━━━┷━━━━━━━━━━━┙
```

“double_grid” draws a grid using double-line box-drawing characters:

```
>>> print(tabulate([["spam", 41.9999], ["eggs", "451.0"]],
...                ["strings", "numbers"], "double_grid"))

 strings      numbers

 spam         41.9999

 eggs        451

```

“fancy_grid” draws a grid using a mix of single and double-line box-drawing characters:

```
>>> print(tabulate([["spam", 41.9999], ["eggs", "451.0"]],
...                ["strings", "numbers"], "fancy_grid"))

│ strings   │   numbers │
```

(continues on next page)

(continued from previous page)

```
│ spam      │   41.9999 │
├───────────┼───────────┤
│ eggs      │  451      │
```

"outline" is the same as the "grid" format but doesn't draw lines between rows:

```
>>> print(tabulate([["spam", 41.9999], ["eggs", "451.0"]],
...                ["strings", "numbers"], "outline"))
+-----------+-----------+
| strings   |   numbers |
+===========+===========+
| spam      |   41.9999 |
| eggs      |  451      |
+-----------+-----------+
```

```
>>> print(tabulate([["spam", 41.9999], ["eggs", "451.0"]], tablefmt="outline"))
+------+----------+
| spam |  41.9999 |
| eggs | 451      |
+------+----------+
```

"simple_outline" is the same as the "simple_grid" format but doesn't draw lines between rows:

```
>>> print(tabulate([["spam", 41.9999], ["eggs", "451.0"]],
...                ["strings", "numbers"], "simple_outline"))
┌───────────┬───────────┐
│ strings   │   numbers │
├───────────┼───────────┤
│ spam      │   41.9999 │
│ eggs      │  451      │
└───────────┴───────────┘
```

"rounded_outline" is the same as the "rounded_grid" format but doesn't draw lines between rows:

```
>>> print(tabulate([["spam", 41.9999], ["eggs", "451.0"]],
...                ["strings", "numbers"], "rounded_outline"))
╭───────────┬───────────╮
│ strings   │   numbers │
├───────────┼───────────┤
│ spam      │   41.9999 │
│ eggs      │  451      │
╰───────────┴───────────╯
```

"heavy_outline" is the same as the "heavy_grid" format but doesn't draw lines between rows:

```
>>> print(tabulate([["spam", 41.9999], ["eggs", "451.0"]],
...                ["strings", "numbers"], "heavy_outline"))
┏━━━━━━━━━━━┳━━━━━━━━━━━┓
┃ strings   ┃   numbers ┃
┣━━━━━━━━━━━╋━━━━━━━━━━━┫
┃ spam      ┃   41.9999 ┃
```

(continues on next page)

(continued from previous page)

```
│ eggs       │   451       │
└────────────┴─────────────┘
```

"mixed_outline" is the same as the "mixed_grid" format but doesn't draw lines between rows:

```
>>> print(tabulate([["spam", 41.9999], ["eggs", "451.0"]],
...                ["strings", "numbers"], "mixed_outline"))
┌────────────┬─────────────┐
│ strings    │   numbers   │
├────────────┼─────────────┤
│ spam       │   41.9999   │
│ eggs       │  451        │
└────────────┴─────────────┘
```

"double_outline" is the same as the "double_grid" format but doesn't draw lines between rows:

```
>>> print(tabulate([["spam", 41.9999], ["eggs", "451.0"]],
...                ["strings", "numbers"], "double_outline"))

 strings      numbers

 spam         41.9999
 eggs        451
```

"fancy_outline" is the same as the "fancy_grid" format but doesn't draw lines between rows:

```
>>> print(tabulate([["spam", 41.9999], ["eggs", "451.0"]],
...                ["strings", "numbers"], "fancy_outline"))

│ strings    │   numbers   │

│ spam       │   41.9999   │
│ eggs       │  451        │
```

"pipe" is like tables in PHP Markdown Extra extension or Pandoc pipe_tables:

```
>>> print(tabulate([["spam", 41.9999], ["eggs", "451.0"]],
...                ["strings", "numbers"], "pipe"))
| strings    |   numbers   |
|:----------|----------:|
| spam       |   41.9999   |
| eggs       |  451        |
```

"presto" is like tables produce by the Presto CLI:

```
>>> print(tabulate([["spam", 41.9999], ["eggs", "451.0"]],
...                ["strings", "numbers"], "presto"))
 strings    |   numbers
-----------+-----------
 spam       |   41.9999
 eggs       |  451
```

```
>>> print(tabulate([["spam", 41.9999], ["eggs", "451.0"]], tablefmt="pipe"))
|:-----|---------:|
| spam |  41.9999 |
| eggs | 451      |
```

"orgtbl" is like tables in Emacs org-mode and orgtbl-mode. They are slightly different from "pipe" format by not using colons to define column alignment, and using a "+" sign to indicate line intersections:

```
>>> print(tabulate([["spam", 41.9999], ["eggs", "451.0"]],
...                ["strings", "numbers"], "orgtbl"))
| strings   |   numbers |
|-----------+-----------|
| spam      |   41.9999 |
| eggs      |  451      |
```

```
>>> print(tabulate([["spam", 41.9999], ["eggs", "451.0"]], tablefmt="orgtbl"))
| spam |  41.9999 |
| eggs | 451      |
```

"rst" is like a simple table format from reStructuredText; please note that reStructuredText accepts also "grid" tables:

```
>>> print(tabulate([["spam", 41.9999], ["eggs", "451.0"]],
...                ["strings", "numbers"], "rst"))
=========  =========
strings      numbers
=========  =========
spam         41.9999
eggs        451
=========  =========
```

```
>>> print(tabulate([["spam", 41.9999], ["eggs", "451.0"]], tablefmt="rst"))
====  ========
spam   41.9999
eggs  451
====  ========
```

"mediawiki" produces a table markup used in Wikipedia and on other MediaWiki-based sites:

```
>>> print(tabulate([["strings", "numbers"], ["spam", 41.9999], ["eggs", "451.0"]],
...                headers="firstrow", tablefmt="mediawiki"))
{| class="wikitable" style="text-align: left;"
|+ <!-- caption -->
|-
! strings   !! align="right"|   numbers
|-
| spam      || align="right"|   41.9999
|-
| eggs      || align="right"|  451
|}
```

"html" produces HTML markup as an html.escape'd str with a ._repr_html_ method so that Jupyter Lab and Notebook display the HTML and a .str property so that the raw HTML remains accessible the unsafehtml table format can be used if an unescaped HTML format is required:

```
>>> print(tabulate([["strings", "numbers"], ["spam", 41.9999], ["eggs", "451.0"]],
...                headers="firstrow", tablefmt="html"))
<table>
<thead>
<tr><th>strings  </th><th style="text-align: right;">  numbers</th></tr>
</thead>
<tbody>
<tr><td>spam     </td><td style="text-align: right;">  41.9999</td></tr>
<tr><td>eggs     </td><td style="text-align: right;">451     </td></tr>
</tbody>
</table>
```

"latex" produces a tabular environment of LaTeX document markup:

```
>>> print(tabulate([["spam", 41.9999], ["eggs", "451.0"]], tablefmt="latex"))
\begin{tabular}{lr}
\hline
 spam &  41.9999 \\
 eggs & 451      \\
\hline
\end{tabular}
```

"latex_raw" is similar to "latex", but doesn't escape special characters, such as backslash and underscore, so LaTeX commands may embedded into cells' values:

```
>>> print(tabulate([["spam$_9$", 41.9999], ["\\emph{eggs}", "451.0"]], tablefmt=
↪"latex_raw"))
\begin{tabular}{lr}
\hline
 spam$_9$    &  41.9999 \\
 \emph{eggs} & 451      \\
\hline
\end{tabular}
```

"latex_booktabs" produces a tabular environment of LaTeX document markup using the booktabs.sty package:

```
>>> print(tabulate([["spam", 41.9999], ["eggs", "451.0"]], tablefmt="latex_booktabs
↪"))
\begin{tabular}{lr}
\toprule
 spam &  41.9999 \\
 eggs & 451      \\
\bottomrule
\end{tabular}
```

"latex_longtable" produces a tabular environment that can stretch along multiple pages, using the longtable package for LaTeX.

```
>>> print(tabulate([["spam", 41.9999], ["eggs", "451.0"]], tablefmt="latex_longtable
↪"))
\begin{longtable}{lr}
\hline
 spam &  41.9999 \\
```

(continues on next page)

(continued from previous page)

```
 eggs & 451      \\
\hline
\end{longtable}
```

3.13.4 Number parsing

By default, anything which can be parsed as a number is a number. This ensures numbers represented as strings are aligned properly. This can lead to weird results for particular strings such as specific git SHAs e.g. "42992e1" will be parsed into the number 429920 and aligned as such.

To completely disable number parsing (and alignment), use *disable_numparse=True*. For more fine grained control, a list column indices is used to disable number parsing only on those columns e.g. *disable_numparse=[0, 2]* would disable number parsing only on the first and third columns.

3.13.5 Column Widths and Auto Line Wrapping

Tabulate will, by default, set the width of each column to the length of the longest element in that column. However, in situations where fields are expected to reasonably be too long to look good as a single line, tabulate can help automate word wrapping long fields for you. Use the parameter *maxcolwidth* to provide a list of maximal column widths

```
>>> print(tabulate(           [('1', 'John Smith',             'This is a rather␣
↪long description that might look better if it is wrapped a bit')],         ␣
↪headers=("Issue Id", "Author", "Description"),           maxcolwidths=[None, None,
↪ 30],           tablefmt="grid"          ))
+------------+------------+-------------------------------+
|   Issue Id | Author     | Description                   |
+============+============+===============================+
|          1 | John Smith | This is a rather long         |
|            |            | description that might look   |
|            |            | better if it is wrapped a bit |
+------------+------------+-------------------------------+
```

Header column width can be specified in a similar way using *maxheadercolwidth*

`xmodaler.utils.memory.`**`retry_if_cuda_oom`**(*func*)

Makes a function retry itself after encountering pytorch's CUDA OOM error. It will first retry after calling *torch.cuda.empty_cache()*.

If that still fails, it will then retry by trying to convert inputs to CPUs. In this case, it expects the function to dispatch to CPU implementation. The return values may become CPU tensors as well and it's user's responsibility to convert it back to CUDA tensor if needed.

Parameters

func – a stateless callable that takes tensor-like objects as arguments

Returns

a callable which retries *func* if OOM is encountered.

Examples:

```
output = retry_if_cuda_oom(some_torch_function)(input1, input2)
# output may be on CPU even if inputs are on GPU
```

Note:

1. When converting inputs to CPU, it will only look at each argument and check if it has *.device* and *.to* for conversion. Nested structures of tensors are not supported.
2. Since the function might be called more than once, it has to be stateless.

class `xmodaler.utils.registry.`**`Registry`**(*name: str*)

Bases: `Iterable[Tuple[str, Any]]`

The registry that provides name -> object mapping, to support third-party users' custom modules.

To create a registry (e.g. a backbone registry):

```
BACKBONE_REGISTRY = Registry('BACKBONE')
```

To register an object:

```
@BACKBONE_REGISTRY.register()
class MyBackbone():
    ...
```

Or:

```
BACKBONE_REGISTRY.register(MyBackbone)
```

`__init__`(*name: str*) → None

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

`get`(*name: str*) → Any

`register`(*obj: Optional[Any] = None*) → Any

Register the given object under the the name *obj.__name__*. Can be used as either a decorator or not. See docstring of this class for usage.

`xmodaler.utils.registry.`**`locate`**(*name: str*) → Any

Locate and return an object `x` using an input string `{x.__module__}.{x.__qualname__}`, such as "module.submodule.class_name".

Raise Exception if it cannot be found.

class `xmodaler.utils.serialize.`**`PicklableWrapper`**(*obj*)

Bases: `object`

Wrap an object to make it more picklable, note that it uses heavy weight serialization libraries that are slower than pickle. It's best to use it only on closures (which are usually not picklable).

This is a simplified version of https://github.com/joblib/joblib/blob/master/joblib/externals/loky/cloudpickle_wrapper.py

`__init__`(*obj*)

PYTHON MODULE INDEX

X

INDEX

Symbols

A

B

F

G

H

I

J

K

L

M

N

P

R

S

T

U

V

W

X