MMF Documentation

Release 1.0.0rc12

Facebook AI Research

Oct 08, 2021

Library Reference

MMF is a modular framework for supercharging vision and language research built on top of PyTorch. Using MMF, researchers and devlopers can train custom models for VQA, Image Captioning, Visual Dialog, Hate Detection and other vision and language tasks.

Read docs for tutorials and documentation.

CHAPTER 1

Citation

If you use MMF in your work or use any models published in MMF, please cite:

```
@misc{singh2020mmf,
author =       {Singh, Amanpreet and Goswami, Vedanuj and Natarajan, Vivek and Jiang,␣
↪Yu and Chen, Xinlei and Shah, Meet and
                Rohrbach, Marcus and Batra, Dhruv and Parikh, Devi},
title =        {MMF: A multimodal framework for vision and language research},
howpublished = {\url{https://github.com/facebookresearch/mmf}},
year =         {2020}
}
```

1.1 common.registry

Registry is central source of truth in MMF. Inspired from Redux's concept of global store, Registry maintains mappings of various information to unique keys. Special functions in registry can be used as decorators to register different kind of classes.

Import the global registry object using

```
from mmf.common.registry import registry
```

Various decorators for registry different kind of classes with unique keys

- Register a trainer: `@registry.register_trainer`
- Register a dataset builder: `@registry.register_builder`
- Register a callback function: `@registry.register_callback`
- Register a metric: `@registry.register_metric`
- Register a loss: `@registry.register_loss`
- Register a fusion technique: `@registery.register_fusion`
- Register a model: `@registry.register_model`

- Register a processor: `@registry.register_processor`
- Register a optimizer: `@registry.register_optimizer`
- Register a scheduler: `@registry.register_scheduler`
- Register a encoder: `@registry.register_encoder`
- Register a decoder: `@registry.register_decoder`
- Register a transformer backend: `@registry.register_transformer_backend`
- Register a transformer head: `@registry.register_transformer_head`
- Register a test reporter: `@registry.register_test_reporter`
- Register a pl datamodule: `@registry.register_datamodule`

class mmf.common.registry.**Registry**

Class for registry object which acts as central source of truth for MMF

classmethod get(*name*, *default=None*, *no_warning=False*)

Get an item from registry with key 'name'

Parameters

- **name** (*string*) – Key whose value needs to be retrieved.
- **default** – If passed and key is not in registry, default value will be returned with a warning. Default: None
- **no_warning** (*bool*) – If passed as True, warning when key doesn't exist will not be generated. Useful for MMF's internal operations. Default: False

Usage:

```
from mmf.common.registry import registry

config = registry.get("config")
```

classmethod register(*name*, *obj*)

Register an item to registry with key 'name'

Parameters **name** – Key with which the item will be registered.

Usage:

```
from mmf.common.registry import registry

registry.register("config", {})
```

classmethod register_builder(*name*)

Register a dataset builder to registry with key 'name'

Parameters **name** – Key with which the metric will be registered.

Usage:

```
from mmf.common.registry import registry
from mmf.datasets.base_dataset_builder import BaseDatasetBuilder


@registry.register_builder("vqa2")
class VQA2Builder(BaseDatasetBuilder):
    ...
```

classmethod register_callback(*name*)
Register a callback to registry with key 'name'

Parameters **name** – Key with which the callback will be registered.

Usage:

```
from mmf.common.registry import registry
from mmf.trainers.callbacks.base import Callback


@registry.register_callback("logistic")
class LogisticCallback(Callback):
    ...
```

classmethod register_datamodule(*name*)
Register a datamodule to registry with key 'name'

Parameters **name** – Key with which the datamodule will be registered.

Usage:

```
from mmf.common.registry import registry
import pytorch_lightning as pl


@registry.register_datamodule("my_datamodule")
class MyDataModule(pl.LightningDataModule):
    ...
```

classmethod register_decoder(*name*)
Register a decoder to registry with key 'name'

Parameters **name** – Key with which the decoder will be registered.

Usage:

```
from mmf.common.registry import registry
from mmf.utils.text import TextDecoder


@registry.register_decoder("nucleus_sampling")
class NucleusSampling(TextDecoder):
    ...
```

classmethod register_encoder(*name*)
Register a encoder to registry with key 'name'

Parameters **name** – Key with which the encoder will be registered.

Usage:

```
from mmf.common.registry import registry
from mmf.modules.encoders import Encoder


@registry.register_encoder("transformer")
class TransformerEncoder(Encoder):
    ...
```

classmethod register_fusion(*name*)
Register a fusion technique to registry with key 'name'

Parameters **name** – Key with which the fusion technique will be registered

Usage:

```
from mmf.common.registry import registry
from torch import nn

@registry.register_fusion("linear_sum")
class LinearSum():
    ...
```

classmethod register_iteration_strategy(*name*)
Register an iteration_strategy to registry with key 'name'

Parameters **name** – Key with which the iteration_strategy will be registered.

Usage:

```
from dataclasses import dataclass
from mmf.common.registry import registry
from mmf.datasets.iterators import IterationStrategy


@registry.register_iteration_strategy("my_iteration_strategy")
class MyStrategy(IterationStrategy):
    @dataclass
    class Config:
        name: str = "my_strategy"
    def __init__(self, config, dataloader):
        ...
```

classmethod register_loss(*name*)
Register a loss to registry with key 'name'

Parameters **name** – Key with which the loss will be registered.

Usage:

```
from mmf.common.registry import registry
from torch import nn

@registry.register_task("logit_bce")
class LogitBCE(nn.Module):
    ...
```

classmethod register_metric(*name*)
Register a metric to registry with key 'name'

Parameters **name** – Key with which the metric will be registered.

Usage:

```
from mmf.common.registry import registry
from mmf.modules.metrics import BaseMetric


@registry.register_metric("r@1")
```

(continues on next page)

(continued from previous page)

```
class RecallAt1(BaseMetric):
    ...
```

classmethod register_model(*name*)
Register a model to registry with key 'name'

Parameters **name** – Key with which the model will be registered.

Usage:

```
from mmf.common.registry import registry
from mmf.models.base_model import BaseModel

@registry.register_task("pythia")
class Pythia(BaseModel):
    ...
```

classmethod register_processor(*name*)
Register a processor to registry with key 'name'

Parameters **name** – Key with which the processor will be registered.

Usage:

```
from mmf.common.registry import registry
from mmf.datasets.processors import BaseProcessor

@registry.register_task("glove")
class GloVe(BaseProcessor):
    ...
```

classmethod register_trainer(*name*)
Register a trainer to registry with key 'name'

Parameters **name** – Key with which the trainer will be registered.

Usage:

```
from mmf.common.registry import registry
from mmf.trainers.custom_trainer import CustomTrainer


@registry.register_trainer("custom_trainer")
class CustomTrainer():
    ...
```

classmethod unregister(*name*)
Remove an item from registry with key 'name'

Parameters **name** – Key which needs to be removed.

Usage:

```
from mmf.common.registry import registry

config = registry.unregister("config")
```

1.2 common.sample

`Sample` and `SampleList` are data structures for arbitrary data returned from a dataset. To work with MMF, minimum requirement for datasets is to return an object of `Sample` class and for models to accept an object of type *SampleList* as an argument.

`Sample` is used to represent an arbitrary sample from dataset, while `SampleList` is list of Sample combined in an efficient way to be used by the model. In simple term, `SampleList` is a batch of Sample but allow easy access of attributes from `Sample` while taking care of properly batching things.

class `mmf.common.sample.`**Sample**(*init_dict=None*)

Sample represent some arbitrary data. All datasets in MMF must return an object of type `Sample`.

Parameters **init_dict** (*Dict*) – Dictionary to init `Sample` class with.

Usage:

```
>>> sample = Sample({"text": torch.tensor(2)})
>>> sample.text.zero_()
# Custom attributes can be added to ``Sample`` after initialization
>>> sample.context = torch.tensor(4)
```

fields()

Get current attributes/fields registered under the sample.

Returns Attributes registered under the Sample.

Return type List[str]

class `mmf.common.sample.`**SampleList**(*samples=None*)

`SampleList` is used to collate a list of `Sample` into a batch during batch preparation. It can be thought of as a merger of list of Dicts into a single Dict.

If `Sample` contains an attribute 'text' of size (2) and there are 10 samples in list, the returned `SampleList` will have an attribute 'text' which is a tensor of size (10, 2).

Parameters **samples** (*type*) – List of `Sample` from which the `SampleList` will be created.

Usage:

```
>>> sample_list = [
        Sample({"text": torch.tensor(2)}),
        Sample({"text": torch.tensor(2)})
    ]
>>> sample_list.text
torch.tensor([2, 2])
```

add_field(*field*, *data*)

Add an attribute `field` with value `data` to the SampleList

Parameters

- **field** (*str*) – Key under which the data will be added.
- **data** (*object*) – Data to be added, can be a `torch.Tensor`, `list` or `Sample`

copy()

Get a copy of the current SampleList

Returns Copy of current SampleList.

Return type *SampleList*

fields()
Get current attributes/fields registered under the SampleList.

Returns list of attributes of the SampleList.

Return type List[str]

get_batch_size()
Get batch size of the current `SampleList`. There must be a tensor be a tensor present inside sample list to use this function. :returns: Size of the batch in `SampleList`. :rtype: int

get_field(*field*)
Get value of a particular attribute

Parameters **field** (*str*) – Attribute whose value is to be returned.

get_fields(*fields*)
Get a new `SampleList` generated from the current `SampleList` but contains only the attributes passed in *fields* argument

Parameters **fields** (*List[str]*) – Attributes whose `SampleList` will be made.

Returns SampleList containing only the attribute values of the fields which were passed.

Return type *SampleList*

get_item_list(*key*)
Get `SampleList` of only one particular attribute that is present in the `SampleList`.

Parameters **key** (*str*) – Attribute whose `SampleList` will be made.

Returns SampleList containing only the attribute value of the key which was passed.

Return type *SampleList*

pin_memory()
In custom batch object, we need to define pin_memory function so that PyTorch can actually apply pinning. This function just individually pins all of the tensor fields

to(*device*, *non_blocking=True*)
Similar to `.to` function on a *torch.Tensor*. Moves all of the tensors present inside the `SampleList` to a particular device. If an attribute's value is not a tensor, it is ignored and kept as it is.

Parameters

- **device** (*str|torch.device*) – Device on which the `SampleList` should moved.
- **non_blocking** (*bool*) – Whether the move should be non_blocking. Default: True

Returns a SampleList moved to the `device`.

Return type *SampleList*

to_dict() → Dict[str, Any]
Converts a sample list to dict, this is useful for TorchScript and for other internal API unification efforts.

Returns A dict representation of current sample list

Return type Dict[str, Any]

mmf.common.sample.**detach_tensor**(*tensor: Any*) → Any
Detaches any element passed which has a *.detach* function defined. Currently, in MMF can be SampleList, Report or a tensor.

Parameters **tensor** (*Any*) – Item to be detached

Returns Detached element

Return type Any

1.3 models.base_model

Models built in MMF need to inherit `BaseModel` class and adhere to a fixed format. To create a model for MMF, follow this quick cheatsheet.

1. Inherit `BaseModel` class, make sure to call `super().__init__()` in your class's `__init__` function.
2. Implement *build* function for your model. If you build everything in `__init__`, you can just return in this function.
3. Write a *forward* function which takes in a `SampleList` as an argument and returns a dict.
4. Register using `@registry.register_model("key")` decorator on top of the class.

If you are doing logits based predictions, the dict you return from your model should contain a *scores* field. Losses are automatically calculated by the `BaseModel` class and added to this dict if not present.

Example:

```
import torch

from mmf.common.registry import registry
from mmf.models.base_model import BaseModel


@registry.register("pythia")
class Pythia(BaseModel):
    # config is model_config from global config
    def __init__(self, config):
        super().__init__(config)

    def build(self):
        ....

    def forward(self, sample_list):
        scores = torch.rand(sample_list.get_batch_size(), 3127)
        return {"scores": scores}
```

class mmf.models.base_model.**BaseModel**(*config: Union[omegaconf.dictconfig.DictConfig, mmf.models.base_model.BaseModel.Config]*)

For integration with MMF's trainer, datasets and other features, models needs to inherit this class, call *super*, write a build function, write a forward function taking a `SampleList` as input and returning a dict as output and finally, register it using `@registry.register_model`

Parameters **config** (*DictConfig*) – `model_config` configuration from global config.

class Config(*model: str = '???', losses: Union[List[mmf.modules.losses.LossConfig], NoneType] = '???'*)

build()

Function to be implemented by the child class, in case they need to build their model separately than `__init__`. All model related downloads should also happen here.

configure_optimizers()

Member function of PL modules. Used only when PL enabled.

format_for_prediction(*results*, *report*)
Implement this method in models if it requires to modify prediction results using report fields. Note that the required fields in report should already be gathered in report.

classmethod format_state_key(*key*)
Can be implemented if something special needs to be done to the key when pretrained model is being loaded. This will adapt and return keys according to that. Useful for backwards compatibility. See updated load_state_dict below. For an example, see VisualBERT model's code.

Parameters **key** (*string*) – key to be formatted

Returns formatted key

Return type string

forward(*sample_list*, **args*, ***kwargs*)
To be implemented by child class. Takes in a `SampleList` and returns back a dict.

Parameters

- **sample_list** (`SampleList`) – SampleList returned by the DataLoader for
- **iteration** (*current*) –

Returns Dict containing scores object.

Return type Dict

init_losses()
Initializes loss for the model based `losses` key. Automatically called by MMF internally after building the model.

load_state_dict(*state_dict*, **args*, ***kwargs*)
Copies parameters and buffers from `state_dict` into this module and its descendants. If `strict` is `True`, then the keys of `state_dict` must exactly match the keys returned by this module's `state_dict()` function.

Parameters

- **state_dict** (*dict*) – a dict containing parameters and persistent buffers.
- **strict** (*bool, optional*) – whether to strictly enforce that the keys in `state_dict` match the keys returned by this module's `state_dict()` function. Default: `True`

Returns

- **missing_keys** is a list of str containing the missing keys
- **unexpected_keys** is a list of str containing the unexpected keys

Return type `NamedTuple` with `missing_keys` and `unexpected_keys` fields

on_load_checkpoint(*checkpoint: Dict[str, Any]*) → None
This is called by the pl.LightningModule before the model's checkpoint is loaded.

on_save_checkpoint(*checkpoint: Dict[str, Any]*) → None
Give the model a chance to add something to the checkpoint. `state_dict` is already there.

Parameters **checkpoint** – A dictionary in which you can save variables to save in a checkpoint. Contents need to be pickleable.

test_step(*batch: mmf.common.sample.SampleList*, *batch_idx: int*, **args*, ***kwargs*)
Member function of PL modules. Used only when PL enabled. To be implemented by child class. Takes in a `SampleList`, batch_idx and returns back a dict.

Parameters

- **sample_list** (SampleList) – SampleList returned by the DataLoader for
- **iteration** (*current*) –

Returns Dict

training_step(*batch: mmf.common.sample.SampleList*, *batch_idx: int*, **args*, ***kwargs*)
Member function of PL modules. Used only when PL enabled. To be implemented by child class. Takes in a SampleList, batch_idx and returns back a dict.

Parameters

- **sample_list** (SampleList) – SampleList returned by the DataLoader for
- **iteration** (*current*) –

Returns Dict containing loss.

Return type Dict

validation_step(*batch: mmf.common.sample.SampleList*, *batch_idx: int*, **args*, ***kwargs*)
Member function of PL modules. Used only when PL enabled. To be implemented by child class. Takes in a SampleList, batch_idx and returns back a dict.

Parameters

- **sample_list** (SampleList) – SampleList returned by the DataLoader for
- **iteration** (*current*) –

Returns Dict

1.4 modules.losses

Losses module contains implementations for various losses used generally in vision and language space. One can register custom losses to be detected by MMF using the following example.

```
from mmf.common.registry import registry
from torch import nn


@registry.register_loss("custom")
class CustomLoss(nn.Module):
    ...
```

Then in your model's config you can specify `losses` attribute to use this loss in the following way:

```
model_config:
    some_model:
        losses:
            - type: custom
            - params: {}
```

class mmf.modules.losses.**AttentionSupervisionLoss**
Loss for attention supervision. Used in case you want to make attentions similar to some particular values.

forward(*sample_list*, *model_output*)
Calculates and returns the multi loss.

Parameters

- **sample_list** (SampleList) – SampleList containing *targets* attribute.
- **model_output** (*Dict*) – Model output containing *scores* attribute.

Returns Float value for loss.

Return type torch.FloatTensor

class mmf.modules.losses.**BCEAndKLLoss**(*weight_softmax*)

binary_cross_entropy_with_logits and kl divergence loss. Calculates both losses and returns a dict with string keys. Similar to bce_kl_combined, but returns both losses.

forward(*sample_list*, *model_output*)

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.

---

class mmf.modules.losses.**BinaryCrossEntropyLoss**

forward(*sample_list*, *model_output*)

Calculates and returns the binary cross entropy.

Parameters

- **sample_list** (SampleList) – SampleList containing *targets* attribute.
- **model_output** (*Dict*) – Model output containing *scores* attribute.

Returns Float value for loss.

Return type torch.FloatTensor

class mmf.modules.losses.**CaptionCrossEntropyLoss**

forward(*sample_list*, *model_output*)

Calculates and returns the cross entropy loss for captions.

Parameters

- **sample_list** (SampleList) – SampleList containing *targets* attribute.
- **model_output** (*Dict*) – Model output containing *scores* attribute.

Returns Float value for loss.

Return type torch.FloatTensor

class mmf.modules.losses.**CombinedLoss**(*weight_softmax*)

forward(*sample_list*, *model_output*)

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.

class mmf.modules.losses.**ContrastiveLoss**

This is a generic contrastive loss typically used for pretraining. No modality assumptions are made here.

forward(*sample_list: Dict[str, torch.Tensor], model_output: Dict[str, torch.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.

class mmf.modules.losses.**CosineEmbeddingLoss**

Cosine embedding loss

forward(*sample_list*, *model_output*)

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.

class mmf.modules.losses.**CrossEntropyLoss**(***params*)

forward(*sample_list*, *model_output*)

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.

class mmf.modules.losses.**InBatchHinge**(*margin: float = 0.0*, *hard: bool = False*)

Based on the code from https://github.com/fartashf/vsepp/blob/master/model.py

forward(*sample_list: Dict[str, torch.Tensor], model_output: Dict[str, torch.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.

class mmf.modules.losses.**LabelSmoothingCrossEntropyLoss**(*label_smoothing=0.1*, *reduction='mean'*, *ignore_index=-100*)

Cross-entropy loss with label smoothing. If *label_smoothing* = 0, then it's canonical cross entropy. The smoothed one-hot encoding is 1 - label_smoothing for true label and label_smoothing / (num_classes - 1) for the rest.

Reference: https://stackoverflow.com/questions/55681502/label-smoothing-in-pytorch

forward(*sample_list*, *model_output*)

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.

---

class mmf.modules.losses.**LogitBinaryCrossEntropy**

Returns Binary Cross Entropy for logits.

Attention: *Key*: logit_bce

forward(*sample_list*, *model_output*)

Calculates and returns the binary cross entropy for logits

Parameters

- **sample_list** (`SampleList`) – SampleList containing *targets* attribute.
- **model_output** (*`Dict`*) – Model output containing *scores* attribute.

Returns Float value for loss.

Return type torch.FloatTensor

class mmf.modules.losses.**LossConfig**(*type: str = '???'*, *params: Dict[str, Any] = '???'*)

class mmf.modules.losses.**Losses**(*loss_list: List[Union[str, mmf.modules.losses.LossConfig]]*)

`Losses` acts as an abstraction for instantiating and calculating losses. `BaseModel` instantiates this class based on the *losses* attribute in the model's configuration *model_config*. `loss_list` needs to be a list for each separate loss containing *type* and *params* attributes.

Parameters **loss_list** (*`ListConfig`*) – Description of parameter *loss_list*.

Example:

```
# losses:
# - type: logit_bce
# Can also contain `params` to specify that particular loss's init params
# - type: combined
config = [{"type": "logit_bce"}, {"type": "combined"}]
losses = Losses(config)
```

Note: Since, `Losses` is instantiated in the `BaseModel`, normal end user mostly doesn't need to use this class.

---

losses
List containing instantiations of each loss passed in config

forward(*sample_list: Dict[str, torch.Tensor], model_output: Dict[str, torch.Tensor]*)
Takes in the original `SampleList` returned from DataLoader and *model_output* returned from the model and returned a Dict containing loss for each of the losses in *losses*.

Parameters

- **sample_list** (`SampleList`) – SampleList given be the dataloader.
- **model_output** (*`Dict`*) – Dict returned from model as output.

Returns Dictionary containing loss value for each of the loss.

Return type Dict

class mmf.modules.losses.**M4CDecodingBCEWithMaskLoss**

forward(*sample_list*, *model_output*)
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.

---

class mmf.modules.losses.**MMFLoss**(*params=None*)
Internal MMF helper and wrapper class for all Loss classes. It makes sure that the value returned from a Loss class is a dict and contain proper dataset type in keys, so that it is easy to figure out which one is the val loss and which one is train loss.

For example: it will return `{"val/vqa2/logit_bce": 27.4}`, in case *logit_bce* is used and SampleList is from *val* set of dataset *vqa2*.

Parameters **params** (*`type`*) – Description of parameter *params*.

Note: Since, `MMFLoss` is used by the `Losses` class, end user doesn't need to worry about it.

---

forward(*sample_list: Dict[str, torch.Tensor], model_output: Dict[str, torch.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.

---

class mmf.modules.losses.**MSELoss**
Mean Squared Error loss

forward(*sample_list*, *model_output*)
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.

class mmf.modules.losses.**MultiLoss**(*params*)

A loss for combining multiple losses with weights.

Parameters **params** (*List(Dict)*) – A list containing parameters for each different loss and their weights.

Example:

```
# MultiLoss works with config like below where each loss's params and
# weights are defined
losses:
- type: multi
  params:
  - type: logit_bce
    weight: 0.3
    params: {}
  - type: attention_supervision
    weight: 0.7
    params: {}
```

forward(*sample_list*, *model_output*, **args*, ***kwargs*)

Calculates and returns the multi loss.

Parameters

- **sample_list** (SampleList) – SampleList containing *attentions* attribute.
- **model_output** (*Dict*) – Model output containing *attention_supervision* attribute.

Returns Float value for loss.

Return type torch.FloatTensor

class mmf.modules.losses.**NLLLoss**

Negative log likelikehood loss.

forward(*sample_list*, *model_output*)

Calculates and returns the negative log likelihood.

Parameters

- **sample_list** (SampleList) – SampleList containing *targets* attribute.
- **model_output** (*Dict*) – Model output containing *scores* attribute.

Returns Float value for loss.

Return type torch.FloatTensor

class mmf.modules.losses.**SoftLabelCrossEntropyLoss**(*ignore_index=-100*, *reduction='mean'*, *normalize_targets=True*)

compute_loss(*targets*, *scores*)

for N examples and C classes - scores: N x C these are raw outputs (without softmax/sigmoid) - targets: N x C or N corresponding targets

Target elements set to ignore_index contribute 0 loss.

Samples where all entries are ignore_index do not contribute to the loss reduction.

forward(*sample_list*, *model_output*)
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.

---

class mmf.modules.losses.**SoftmaxKlDivLoss**

forward(*sample_list*, *model_output*)
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.

---

class mmf.modules.losses.**TripleLogitBinaryCrossEntropy**
This is used for Three-branch fusion only. We predict scores and compute cross entropy loss for each of branches.

forward(*sample_list*, *model_output*)
Calculates and returns the binary cross entropy for logits :param sample_list: SampleList containing *targets* attribute. :type sample_list: SampleList :param model_output: Model output containing *scores* attribute. :type model_output: Dict

Returns Float value for loss.

Return type torch.FloatTensor

class mmf.modules.losses.**WeightedSoftmaxLoss**

forward(*sample_list*, *model_output*)
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.

---

class mmf.modules.losses.**WrongLoss**

forward(*sample_list*, *model_output*)
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.

1.5 modules.metrics

The metrics module contains implementations of various metrics used commonly to understand how well our models are performing. For e.g. accuracy, vqa_accuracy, r@1 etc.

For implementing your own metric, you need to follow these steps:

1. Create your own metric class and inherit `BaseMetric` class.
2. In the `__init__` function of your class, make sure to call `super().__init__('name')` where 'name' is the name of your metric. If you require any parameters in your `__init__` function, you can use keyword arguments to represent them and metric constructor will take care of providing them to your class from config.
3. Implement a `calculate` function which takes in `SampleList` and *model_output* as input and return back a float tensor/number.
4. Register your metric with a key 'name' by using decorator, `@registry.register_metric('name')`.

Example:

```
import torch

from mmf.common.registry import registry
from mmf.modules.metrics import BaseMetric

@registry.register_metric("some")
class SomeMetric(BaseMetric):
    def __init__(self, some_param=None):
        super().__init__("some")
        ....

    def calculate(self, sample_list, model_output):
        metric = torch.tensor(2, dtype=torch.float)
        return metric
```

Example config for above metric:

```
model_config:
    pythia:
        metrics:
        - type: some
          params:
            some_param: a
```

class mmf.modules.metrics.**Accuracy**(*score_key='scores'*, *target_key='targets'*, *topk=1*)

Metric for calculating accuracy.

Key: `accuracy`

calculate(*sample_list*, *model_output*, **args*, ***kwargs*)

Calculate accuracy and return it back.

Parameters

- **sample_list** (SampleList) – SampleList provided by DataLoader for current iteration
- **model_output** (*Dict*) – Dict returned by model.

Returns accuracy.

Return type torch.FloatTensor

class mmf.modules.metrics.**AveragePrecision**(**args*, ***kwargs*)

Metric for calculating Average Precision. See more details at sklearn.metrics.average_precision_score # noqa If you are looking for binary case, please take a look at binary_ap **Key:** ap

calculate(*sample_list*, *model_output*, **args*, ***kwargs*)

Calculate AP and returns it back. The function performs softmax on the logits provided and then calculated the AP.

Parameters

- **sample_list** (SampleList) – SampleList provided by DataLoader for current iteration.
- **model_output** (*Dict*) – Dict returned by model. This should contain "scores" field pointing to logits returned from the model.

Returns AP.

Return type torch.FloatTensor

class mmf.modules.metrics.**BaseMetric**(*name*, **args*, ***kwargs*)

Base class to be inherited by all metrics registered to MMF. See the description on top of the file for more information. Child class must implement `calculate` function.

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

calculate(*sample_list*, *model_output*, **args*, ***kwargs*)

Abstract method to be implemented by the child class. Takes in a `SampleList` and a dict returned by model as output and returns back a float tensor/number indicating value for this metric.

Parameters

- **sample_list** (SampleList) – SampleList provided by the dataloader for the current iteration.
- **model_output** (*Dict*) – Output dict from the model for the current SampleList

Returns Value of the metric.

Return type torch.Tensor|float

class mmf.modules.metrics.**BinaryAP**(**args*, ***kwargs*)

Metric for calculating Binary Average Precision. See more details at sklearn.metrics.average_precision_score # noqa **Key:** `binary_ap`

calculate(*sample_list*, *model_output*, **args*, ***kwargs*)

Calculate Binary AP and returns it back. The function performs softmax on the logits provided and then calculated the binary AP.

Parameters

- **sample_list** (SampleList) – SampleList provided by DataLoader for current iteration.
- **model_output** (*Dict*) – Dict returned by model. This should contain "scores" field pointing to logits returned from the model.

Returns AP.

Return type torch.FloatTensor

class mmf.modules.metrics.**BinaryF1**(**args*, ***kwargs*)
Metric for calculating Binary F1.

Key: `binary_f1`

class mmf.modules.metrics.**BinaryF1PrecisionRecall**(**args*, ***kwargs*)
Metric for calculating Binary F1 Precision and Recall.

Key: `binary_f1_precision_recall`

class mmf.modules.metrics.**CaptionBleu4Metric**
Metric for calculating caption accuracy using BLEU4 Score.

Key: `caption_bleu4`

calculate(*sample_list*, *model_output*, **args*, ***kwargs*)
Calculate accuracy and return it back.

Parameters

- **sample_list** (`SampleList`) – SampleList provided by DataLoader for current iteration
- **model_output** (*Dict*) – Dict returned by model.

Returns bleu4 score.

Return type torch.FloatTensor

class mmf.modules.metrics.**DetectionMeanAP**(*dataset_json_files*, **args*, ***kwargs*)
Metric for calculating the detection mean average precision (mAP) using the COCO evaluation toolkit, returning the default COCO-style mAP@IoU=0.50:0.95

Key: `detection_mean_ap`

calculate(*sample_list*, *model_output*, *execute_on_master_only=True*, **args*, ***kwargs*)
Calculate detection mean AP (mAP) from the prediction list and the dataset annotations. The function returns COCO-style mAP@IoU=0.50:0.95.

Parameters

- **sample_list** (`SampleList`) – SampleList provided by DataLoader for current iteration.
- **model_output** (*Dict*) – Dict returned by model. This should contain "prediction_report" field, which is a list of detection predictions from the model.
- **execute_on_master_only** (*bool*) – Whether to only run mAP evaluation on the master node over the gathered detection prediction (to avoid wasting computation and CPU OOM). Default: True (only run mAP evaluation on master).

Returns COCO-style mAP@IoU=0.50:0.95.

Return type torch.FloatTensor

class mmf.modules.metrics.**F1**(**args*, ***kwargs*)
Metric for calculating F1. Can be used with type and params argument for customization. params will be directly passed to sklearn f1 function. **Key:** `f1`

calculate(*sample_list*, *model_output*, **args*, ***kwargs*)
Calculate f1 and return it back.

Parameters

- **sample_list** (SampleList) – SampleList provided by DataLoader for current iteration
- **model_output** (*Dict*) – Dict returned by model.

Returns f1.

Return type torch.FloatTensor

class mmf.modules.metrics.**F1PrecisionRecall**(**args*, ***kwargs*)

Metric for calculating F1 precision and recall. params will be directly passed to sklearn precision_recall_fscore_support function. **Key:** f1_precision_recall

calculate(*sample_list*, *model_output*, **args*, ***kwargs*)

Calculate f1_precision_recall and return it back as a dict.

Parameters

- **sample_list** (SampleList) – SampleList provided by DataLoader for current iteration
- **model_output** (*Dict*) – Dict returned by model.

Returns

Dict('f1': torch.FloatTensor, 'precision': torch.FloatTensor, 'recall': torch.FloatTensor

)

class mmf.modules.metrics.**MacroAP**(**args*, ***kwargs*)

Metric for calculating Macro Average Precision.

Key: macro_ap

class mmf.modules.metrics.**MacroF1**(**args*, ***kwargs*)

Metric for calculating Macro F1.

Key: macro_f1

class mmf.modules.metrics.**MacroF1PrecisionRecall**(**args*, ***kwargs*)

Metric for calculating Macro F1 Precision and Recall.

Key: macro_f1_precision_recall

class mmf.modules.metrics.**MacroROC_AUC**(**args*, ***kwargs*)

Metric for calculating Macro ROC_AUC.

Key: macro_roc_auc

class mmf.modules.metrics.**MeanRank**

Calculate MeanRank which specifies what was the average rank of the chosen candidate.

Key: mean_r.

calculate(*sample_list*, *model_output*, **args*, ***kwargs*)

Calculate Mean Rank and return it back.

Parameters

- **sample_list** (SampleList) – SampleList provided by DataLoader for current iteration
- **model_output** (*Dict*) – Dict returned by model.

Returns mean rank

Return type torch.FloatTensor

class mmf.modules.metrics.**MeanReciprocalRank**
Calculate reciprocal of mean rank..

Key: mean_rr.

calculate(*sample_list*, *model_output*, **args*, ***kwargs*)
Calculate Mean Reciprocal Rank and return it back.

Parameters

- **sample_list** (SampleList) – SampleList provided by DataLoader for current iteration
- **model_output** (*Dict*) – Dict returned by model.

Returns Mean Reciprocal Rank

Return type torch.FloatTensor

class mmf.modules.metrics.**Metrics**(*metric_list*)
Internally used by MMF, Metrics acts as wrapper for handling calculation of metrics over various metrics specified by the model in the config. It initializes all of the metrics and when called it runs calculate on each of them one by one and returns back a dict with proper naming back. For e.g. an example dict returned by Metrics class: `{'val/vqa_accuracy': 0.3, 'val/r@1': 0.8}`

Parameters **metric_list** (*ListConfig*) – List of DictConfigs where each DictConfig specifies name and parameters of the metrics used.

class mmf.modules.metrics.**MicroAP**(**args*, ***kwargs*)
Metric for calculating Micro Average Precision.

Key: micro_ap

class mmf.modules.metrics.**MicroF1**(**args*, ***kwargs*)
Metric for calculating Micro F1.

Key: micro_f1

class mmf.modules.metrics.**MicroF1PrecisionRecall**(**args*, ***kwargs*)
Metric for calculating Micro F1 Precision and Recall.

Key: micro_f1_precision_recall

class mmf.modules.metrics.**MicroROC_AUC**(**args*, ***kwargs*)
Metric for calculating Micro ROC_AUC.

Key: micro_roc_auc

class mmf.modules.metrics.**MultiLabelF1**(**args*, ***kwargs*)
Metric for calculating Multilabel F1.

Key: multilabel_f1

class mmf.modules.metrics.**MultiLabelMacroF1**(**args*, ***kwargs*)
Metric for calculating Multilabel Macro F1.

Key: multilabel_macro_f1

class mmf.modules.metrics.**MultiLabelMicroF1**(**args*, ***kwargs*)
Metric for calculating Multilabel Micro F1.

Key: multilabel_micro_f1

class mmf.modules.metrics.**OCRVQAAccuracy**

class mmf.modules.metrics.**ROC_AUC**(**args*, ***kwargs*)
Metric for calculating ROC_AUC. See more details at sklearn.metrics.roc_auc_score # noqa

Note: ROC_AUC is not defined when expected tensor only contains one label. Make sure you have both labels always or use it on full val only

Key: roc_auc

calculate(*sample_list*, *model_output*, **args*, ***kwargs*)
Calculate ROC_AUC and returns it back. The function performs softmax on the logits provided and then calculated the ROC_AUC.

Parameters

- **sample_list** (SampleList) – SampleList provided by DataLoader for current iteration.
- **model_output** (*Dict*) – Dict returned by model. This should contain "scores" field pointing to logits returned from the model.

Returns ROC_AUC.

Return type torch.FloatTensor

class mmf.modules.metrics.**RecallAt1**
Calculate Recall@1 which specifies how many time the chosen candidate was rank 1.

Key: r@1.

calculate(*sample_list*, *model_output*, **args*, ***kwargs*)
Calculate Recall@1 and return it back.

Parameters

- **sample_list** (SampleList) – SampleList provided by DataLoader for current iteration
- **model_output** (*Dict*) – Dict returned by model.

Returns Recall@1

Return type torch.FloatTensor

class mmf.modules.metrics.**RecallAt10**
Calculate Recall@10 which specifies how many time the chosen candidate was among first 10 ranks.

Key: r@10.

calculate(*sample_list*, *model_output*, **args*, ***kwargs*)
Calculate Recall@10 and return it back.

Parameters

- **sample_list** (SampleList) – SampleList provided by DataLoader for current iteration
- **model_output** (*Dict*) – Dict returned by model.

Returns Recall@10

Return type torch.FloatTensor

class mmf.modules.metrics.**RecallAt10_ret**

calculate(*sample_list: Dict[str, torch.Tensor], model_output: Dict[str, torch.Tensor], *args, **kwargs*)

Abstract method to be implemented by the child class. Takes in a `SampleList` and a dict returned by model as output and returns back a float tensor/number indicating value for this metric.

Parameters

- **sample_list** (`SampleList`) – SampleList provided by the dataloader for the current iteration.
- **model_output** (*Dict*) – Output dict from the model for the current SampleList

Returns Value of the metric.

Return type torch.Tensor|float

class mmf.modules.metrics.**RecallAt10_rev_ret**

calculate(*sample_list: Dict[str, torch.Tensor], model_output: Dict[str, torch.Tensor], *args, **kwargs*)

Abstract method to be implemented by the child class. Takes in a `SampleList` and a dict returned by model as output and returns back a float tensor/number indicating value for this metric.

Parameters

- **sample_list** (`SampleList`) – SampleList provided by the dataloader for the current iteration.
- **model_output** (*Dict*) – Output dict from the model for the current SampleList

Returns Value of the metric.

Return type torch.Tensor|float

class mmf.modules.metrics.**RecallAt1_ret**

calculate(*sample_list: Dict[str, torch.Tensor], model_output: Dict[str, torch.Tensor], *args, **kwargs*)

Abstract method to be implemented by the child class. Takes in a `SampleList` and a dict returned by model as output and returns back a float tensor/number indicating value for this metric.

Parameters

- **sample_list** (`SampleList`) – SampleList provided by the dataloader for the current iteration.
- **model_output** (*Dict*) – Output dict from the model for the current SampleList

Returns Value of the metric.

Return type torch.Tensor|float

class mmf.modules.metrics.**RecallAt1_rev_ret**

calculate(*sample_list: Dict[str, torch.Tensor], model_output: Dict[str, torch.Tensor], *args, **kwargs*)

Abstract method to be implemented by the child class. Takes in a `SampleList` and a dict returned by model as output and returns back a float tensor/number indicating value for this metric.

Parameters

- **sample_list** (`SampleList`) – SampleList provided by the dataloader for the current iteration.

- **model_output** (*Dict*) – Output dict from the model for the current SampleList

Returns Value of the metric.

Return type torch.Tensor|float

class mmf.modules.metrics.**RecallAt5**

Calculate Recall@5 which specifies how many time the chosen candidate was among first 5 rank.

Key: `r@5`.

calculate(*sample_list*, *model_output*, **args*, ***kwargs*)

Calculate Recall@5 and return it back.

Parameters

- **sample_list** (SampleList) – SampleList provided by DataLoader for current iteration
- **model_output** (*Dict*) – Dict returned by model.

Returns Recall@5

Return type torch.FloatTensor

class mmf.modules.metrics.**RecallAt5_ret**

calculate(*sample_list: Dict[str, torch.Tensor]*, *model_output: Dict[str, torch.Tensor]*, **args*, ***kwargs*)

Abstract method to be implemented by the child class. Takes in a `SampleList` and a dict returned by model as output and returns back a float tensor/number indicating value for this metric.

Parameters

- **sample_list** (SampleList) – SampleList provided by the dataloader for the current iteration.
- **model_output** (*Dict*) – Output dict from the model for the current SampleList

Returns Value of the metric.

Return type torch.Tensor|float

class mmf.modules.metrics.**RecallAt5_rev_ret**

calculate(*sample_list: Dict[str, torch.Tensor]*, *model_output: Dict[str, torch.Tensor]*, **args*, ***kwargs*)

Abstract method to be implemented by the child class. Takes in a `SampleList` and a dict returned by model as output and returns back a float tensor/number indicating value for this metric.

Parameters

- **sample_list** (SampleList) – SampleList provided by the dataloader for the current iteration.
- **model_output** (*Dict*) – Output dict from the model for the current SampleList

Returns Value of the metric.

Return type torch.Tensor|float

class mmf.modules.metrics.**RecallAtK**(*name='recall@k'*)

calculate(*sample_list*, *model_output*, *k*, **args*, ***kwargs*)
Abstract method to be implemented by the child class. Takes in a `SampleList` and a dict returned by model as output and returns back a float tensor/number indicating value for this metric.

Parameters

- **sample_list** (`SampleList`) – SampleList provided by the dataloader for the current iteration.
- **model_output** (*Dict*) – Output dict from the model for the current SampleList

Returns Value of the metric.

Return type torch.Tensor|float

class mmf.modules.metrics.**RecallAtK_ret**(*name='recall@k'*)

calculate(*sample_list: Dict[str, torch.Tensor], model_output: Dict[str, torch.Tensor], k: int, flip=False, *args, **kwargs*)
Abstract method to be implemented by the child class. Takes in a `SampleList` and a dict returned by model as output and returns back a float tensor/number indicating value for this metric.

Parameters

- **sample_list** (`SampleList`) – SampleList provided by the dataloader for the current iteration.
- **model_output** (*Dict*) – Output dict from the model for the current SampleList

Returns Value of the metric.

Return type torch.Tensor|float

class mmf.modules.metrics.**RecallAtPrecisionK**(*p_threshold*, **args*, ***kwargs*)
Metric for calculating recall when precision is above a particular threshold. Use *p_threshold* param to specify the precision threshold i.e. k. Accepts precision in both 0-1 and 1-100 format.

Key: `r@pk`

calculate(*sample_list*, *model_output*, **args*, ***kwargs*)
Calculate Recall at precision k and returns it back. The function performs softmax on the logits provided and then calculated the metric.

Parameters

- **sample_list** (`SampleList`) – SampleList provided by DataLoader for current iteration.
- **model_output** (*Dict*) – Dict returned by model. This should contain "scores" field pointing to logits returned from the model.

Returns Recall @ precision k.

Return type torch.FloatTensor

class mmf.modules.metrics.**STVQAANLS**

class mmf.modules.metrics.**STVQAAccuracy**

class mmf.modules.metrics.**TextCapsBleu4**

class mmf.modules.metrics.**TextVQAAccuracy**

calculate(*sample_list*, *model_output*, **args*, ***kwargs*)
Abstract method to be implemented by the child class. Takes in a `SampleList` and a dict returned by model as output and returns back a float tensor/number indicating value for this metric.

Parameters

- **sample_list** (`SampleList`) – SampleList provided by the dataloader for the current iteration.
- **model_output** (*Dict*) – Output dict from the model for the current SampleList

Returns Value of the metric.

Return type torch.Tensor|float

class mmf.modules.metrics.**TopKAccuracy**(*score_key: str*, *k: int*)

class mmf.modules.metrics.**VQAAccuracy**
Calculate VQAAccuracy. Find more information here

Key: `vqa_accuracy`.

calculate(*sample_list*, *model_output*, **args*, ***kwargs*)
Calculate vqa accuracy and return it back.

Parameters

- **sample_list** (`SampleList`) – SampleList provided by DataLoader for current iteration
- **model_output** (*Dict*) – Dict returned by model.

Returns VQA Accuracy

Return type torch.FloatTensor

class mmf.modules.metrics.**VQAEvalAIAccuracy**
Calculate Eval AI VQAAccuracy. Find more information here This is more accurate and similar comparision to Eval AI but is slower compared to vqa_accuracy.

Key: `vqa_evalai_accuracy`.

calculate(*sample_list*, *model_output*, **args*, ***kwargs*)
Calculate vqa accuracy and return it back.

Parameters

- **sample_list** (`SampleList`) – SampleList provided by DataLoader for current iteration
- **model_output** (*Dict*) – Dict returned by model.

Returns VQA Accuracy

Return type torch.FloatTensor

1.6 datasets.base_dataset_builder

In MMF, for adding new datasets, dataset builder for datasets need to be added. A new dataset builder must inherit `BaseDatasetBuilder` class and implement `load` and `build` functions.

`build` is used to build a dataset when it is not available. For e.g. downloading the ImDBs for a dataset. In future, we plan to add a `build` to add dataset builder to ease setup of MMF.

`load` is used to load a dataset from specific path. `load` needs to return an instance of subclass of `mmf.datasets.base_dataset.BaseDataset`.

See complete example for `VQA2DatasetBuilder` here.

Example:

```
from torch.utils.data import Dataset

from mmf.datasets.base_dataset_builder import BaseDatasetBuilder
from mmf.common.registry import registry

@registry.register_builder("my")
class MyBuilder(BaseDatasetBuilder):
    def __init__(self):
        super().__init__("my")

    def load(self, config, dataset_type, *args, **kwargs):
        ...
        return Dataset()

    def build(self, config, dataset_type, *args, **kwargs):
        ...
```

class mmf.datasets.base_dataset_builder.**BaseDatasetBuilder**(*dataset_name: Optional[str] = None*, **args*, ***kwargs*)

Base class for implementing dataset builders. See more information on top. Child class needs to implement `build` and `load`.

Parameters **dataset_name** (*str*) – Name of the dataset passed from child.

build(*config*, *dataset_type='train'*, **args*, ***kwargs*)

This is used to build a dataset first time. Implement this method in your child dataset builder class.

Parameters

- **config** (*DictConfig*) – Configuration of this dataset loaded from config.
- **dataset_type** (*str*) – Type of dataset, train|val|test

build_dataset(*config*, *dataset_type='train'*, **args*, ***kwargs*)

Similar to load function, used by MMF to build a dataset for first time when it is not available. This internally calls 'build' function. Override that function in your child class.

NOTE: The caller to this function should only call this on master process in a distributed settings so that downloads and build only happen on master process and others can just load it. Make sure to call synchronize afterwards to bring all processes in sync.

Parameters

- **config** (*DictConfig*) – Configuration of this dataset loaded from config.
- **dataset_type** (*str*) – Type of dataset, train|val|test

Warning: DO NOT OVERRIDE in child class. Instead override `build`.

load(*config*, *dataset_type='train'*, **args*, ***kwargs*)

This is used to prepare the dataset and load it from a path. Override this method in your child dataset builder class.

Parameters

- **config** (*DictConfig*) – Configuration of this dataset loaded from config.
- **dataset_type** (*str*) – Type of dataset, train|val|test

Returns Dataset containing data to be trained on

Return type dataset (*BaseDataset*)

load_dataset(*config*, *dataset_type='train'*, **args*, ***kwargs*)

Main load function use by MMF. This will internally call `load` function. Calls `init_processors` and `try_fast_read` on the dataset returned from `load`

Parameters

- **config** (*DictConfig*) – Configuration of this dataset loaded from config.
- **dataset_type** (*str*) – Type of dataset, train|val|test

Returns Dataset containing data to be trained on

Return type dataset (*BaseDataset*)

> **Warning:** DO NOT OVERRIDE in child class. Instead override `load`.

prepare_data(*config*, **args*, ***kwargs*)

NOTE: The caller to this function should only call this on master process in a distributed settings so that downloads and build only happen on master process and others can just load it. Make sure to call synchronize afterwards to bring all processes in sync.

Lightning automatically wraps datamodule in a way that it is only called on a master node, but for extra precaution as lightning can introduce bugs, we should always call this under master process with extra checks on our sides as well.

setup(*stage: Optional[str] = None*, *config: Optional[omegaconf.dictconfig.DictConfig] = None*)

Called at the beginning of fit (train + validate), validate, test, and predict. This is a good hook when you need to build models dynamically or adjust something about them. This hook is called on every process when using DDP.

Parameters **stage** – either `'fit'`, `'validate'`, `'test'`, or `'predict'`

Example:

```
class LitModel(...):
    def __init__(self):
        self.l1 = None

    def prepare_data(self):
        download_data()
        tokenize()

        # don't do this
        self.something = else

    def setup(stage):
        data = Load_data(...)
        self.l1 = nn.Linear(28, data.num_classes)
```

teardown(**args*, ***kwargs*) → None

Called at the end of fit (train + validate), validate, test, predict, or tune.

Parameters **stage** – either `'fit'`, `'validate'`, `'test'`, or `'predict'`

test_dataloader(**args*, ***kwargs*)

Implement one or multiple PyTorch DataLoaders for testing.

The dataloader you return will not be reloaded unless you set **:paramref:'~pytorch_lightning.trainer.Trainer.reload_dataloaders_every_n_epochs'** to a postive integer.

For data processing use the following pattern:

- download in *prepare_data()*
- process and split in *setup()*

However, the above are only necessary for distributed processing.

Warning: do not assign state in prepare_data

- `fit()`
- ...
- *prepare_data()*
- *setup()*
- *train_dataloader()*
- *val_dataloader()*
- *test_dataloader()*

Note: Lightning adds the correct sampler for distributed and arbitrary hardware. There is no need to set it yourself.

Returns A `torch.utils.data.DataLoader` or a sequence of them specifying testing samples.

Example:

```
def test_dataloader(self):
    transform = transforms.Compose([transforms.ToTensor(),
                                    transforms.Normalize((0.5,), (1.0,))])
    dataset = MNIST(root='/path/to/mnist/', train=False, transform=transform,
                    download=True)
    loader = torch.utils.data.DataLoader(
        dataset=dataset,
        batch_size=self.batch_size,
        shuffle=False
    )

    return loader

# can also return multiple dataloaders
def test_dataloader(self):
    return [loader_a, loader_b, ..., loader_n]
```

Note: If you don't need a test dataset and a `test_step()`, you don't need to implement this method.

Note: In the case where you return multiple test dataloaders, the `test_step()` will have an argument `dataloader_idx` which matches the order here.

train_dataloader(**args*, ***kwargs*)

Implement one or more PyTorch DataLoaders for training.

Returns A collection of `torch.utils.data.DataLoader` specifying training samples. In the case of multiple dataloaders, please see this page.

The dataloader you return will not be reloaded unless you set **:paramref:'~pytorch_lightning.trainer.Trainer.reload_dataloaders_every_n_epochs'** to a positive integer.

For data processing use the following pattern:

- download in *prepare_data()*
- process and split in *setup()*

However, the above are only necessary for distributed processing.

Warning: do not assign state in prepare_data

- `fit()`
- ...
- *prepare_data()*
- *setup()*
- *train_dataloader()*

Note: Lightning adds the correct sampler for distributed and arbitrary hardware. There is no need to set it yourself.

Example:

```
# single dataloader
def train_dataloader(self):
    transform = transforms.Compose([transforms.ToTensor(),
                                    transforms.Normalize((0.5,), (1.0,))])
    dataset = MNIST(root='/path/to/mnist/', train=True, transform=transform,
                    download=True)
    loader = torch.utils.data.DataLoader(
        dataset=dataset,
        batch_size=self.batch_size,
        shuffle=True
    )
    return loader

# multiple dataloaders, return as list
def train_dataloader(self):
```

(continues on next page)

(continued from previous page)

```
    mnist = MNIST(...)
    cifar = CIFAR(...)
    mnist_loader = torch.utils.data.DataLoader(
        dataset=mnist, batch_size=self.batch_size, shuffle=True
    )
    cifar_loader = torch.utils.data.DataLoader(
        dataset=cifar, batch_size=self.batch_size, shuffle=True
    )
    # each batch will be a list of tensors: [batch_mnist, batch_cifar]
    return [mnist_loader, cifar_loader]

# multiple dataloader, return as dict
def train_dataloader(self):
    mnist = MNIST(...)
    cifar = CIFAR(...)
    mnist_loader = torch.utils.data.DataLoader(
        dataset=mnist, batch_size=self.batch_size, shuffle=True
    )
    cifar_loader = torch.utils.data.DataLoader(
        dataset=cifar, batch_size=self.batch_size, shuffle=True
    )
    # each batch will be a dict of tensors: {'mnist': batch_mnist, 'cifar':␣
↪batch_cifar}
    return {'mnist': mnist_loader, 'cifar': cifar_loader}
```

val_dataloader(**args*, ***kwargs*)

Implement one or multiple PyTorch DataLoaders for validation.

The dataloader you return will not be reloaded unless you set **:param-ref:'~pytorch_lightning.trainer.Trainer.reload_dataloaders_every_n_epochs'** to a positive integer.

It's recommended that all data downloads and preparation happen in *prepare_data()*.

- fit()
- ...
- *prepare_data()*
- *train_dataloader()*
- *val_dataloader()*
- *test_dataloader()*

Note: Lightning adds the correct sampler for distributed and arbitrary hardware There is no need to set it yourself.

Returns A `torch.utils.data.DataLoader` or a sequence of them specifying validation samples.

Examples:

```
def val_dataloader(self):
    transform = transforms.Compose([transforms.ToTensor(),
                                    transforms.Normalize((0.5,), (1.0,))])
    dataset = MNIST(root='/path/to/mnist/', train=False,
```

(continues on next page)

(continued from previous page)

```
                    transform=transform, download=True)
    loader = torch.utils.data.DataLoader(
        dataset=dataset,
        batch_size=self.batch_size,
        shuffle=False
    )

    return loader

# can also return multiple dataloaders
def val_dataloader(self):
    return [loader_a, loader_b, ..., loader_n]
```

Note: If you don't need a validation dataset and a `validation_step()`, you don't need to implement this method.

Note: In the case where you return multiple validation dataloaders, the `validation_step()` will have an argument `dataloader_idx` which matches the order here.

1.7 datasets.base_dataset

class mmf.datasets.base_dataset.**BaseDataset**(*dataset_name*, *config*, *dataset_type='train'*, **args*, ***kwargs*)

Base class for implementing a dataset. Inherits from PyTorch's Dataset class but adds some custom functionality on top. Processors mentioned in the configuration are automatically initialized for the end user.

Parameters

- **dataset_name** (*str*) – Name of your dataset to be used a representative in text strings
- **dataset_type** (*str*) – Type of your dataset. Normally, train|val|test
- **config** (*DictConfig*) – Configuration for the current dataset

load_item(*idx*)

Implement if you need to separately load the item and cache it.

Parameters **idx** (*int*) – Index of the sample to be loaded.

prepare_batch(*batch*)

Can be possibly overridden in your child class. Not supported w Lightning trainer

Prepare batch for passing to model. Whatever returned from here will be directly passed to model's forward function. Currently moves the batch to proper device.

Parameters **batch** (SampleList) – sample list containing the currently loaded batch

Returns

Returns a sample representing current batch loaded

Return type sample_list (*SampleList*)

1.8 datasets.processors

The processors exist in MMF to make data processing pipelines in various datasets as similar as possible while allowing code reuse.

The processors also help maintain proper abstractions to keep only what matters inside the dataset's code. This allows us to keep the dataset `__getitem__` logic really clean and no need about maintaining opinions about data type. Processors can work on both images and text due to their generic structure.

To create a new processor, follow these steps:

1. Inherit the `BaseProcessor` class.
2. Implement `_call` function which takes in a dict and returns a dict with same keys preprocessed as well as any extra keys that need to be returned.
3. Register the processor using `@registry.register_processor('name')` to registry where 'name' will be used to refer to your processor later.

In processor's config you can specify `preprocessor` option to specify different kind of preprocessors you want in your dataset.

Let's break down processor's config inside a dataset (VQA2.0) a bit to understand different moving parts.

Config:

```
dataset_config:
  vqa2:
    data_dir: ${env.data_dir}
    processors:
      text_processor:
        type: vocab
        params:
          max_length: 14
          vocab:
            type: intersected
            embedding_name: glove.6B.300d
            vocab_file: vqa2/defaults/extras/vocabs/vocabulary_100k.txt
          preprocessor:
            type: simple_sentence
            params: {}
```

`BaseDataset` will init the processors and they will available inside your dataset with same attribute name as the key name, for e.g. *text_processor* will be available as *self.text_processor* inside your dataset. As is with every module in MMF, processor also accept a `DictConfig` with a *type* and *params* attributes. *params* defined the custom parameters for each of the processors. By default, processor initialization process will also init *preprocessor* attribute which can be a processor config in itself. *preprocessor* can be then be accessed inside the processor's functions.

Example:

```
from mmf.common.registry import registry
from mmf.datasets.processors import BaseProcessor

@registry.register_processor('my_processor')
class MyProcessor(BaseProcessor):
    def __init__(self, config, *args, **kwargs):
        return

    def __call__(self, item, *args, **kwargs):
```

(continues on next page)

(continued from previous page)

```
        text = item['text']
        text = [t.strip() for t in text.split(" ")]
        return {"text": text}
```

class mmf.datasets.processors.processors.**BBoxProcessor**(*config*, **args*, ***kwargs*)

Generates bboxes in proper format. Takes in a dict which contains "info" key which is a list of dicts containing following for each of the the bounding box

Example bbox input:

```
{
    "info": [
        {
            "bounding_box": {
                "top_left_x": 100,
                "top_left_y": 100,
                "width": 200,
                "height": 300
            }
        },
        ...
    ]
}
```

This will further return a Sample in a dict with key "bbox" with last dimension of 4 corresponding to "xyxy". So sample will look like following:

Example Sample:

```
Sample({
    "coordinates": torch.Size(n, 4),
    "width": List[number], # size n
    "height": List[number], # size n
    "bbox_types": List[str] # size n, either xyxy or xywh.
    # currently only supports xyxy.
})
```

class mmf.datasets.processors.processors.**BaseProcessor**(**args*, *config: Optional[omegaconf.dictconfig.DictConfig] = None*, ***kwargs*)

Every processor in MMF needs to inherit this class for compatibility with MMF. End user mainly needs to implement __call__ function.

Parameters **config** (*DictConfig*) – Config for this processor, containing *type* and *params* attributes if available.

class mmf.datasets.processors.processors.**BatchProcessor**(*config: mmf.datasets.processors.processors.BatchProcesso*
**args*, ***kwargs*)

BatchProcessor is an extension of normal processor which usually are used in cases where dataset works on full batch instead of samples. Such cases can be observed in the case of the iterable datasets. BatchProcessor if provided with processors key in the config, will initialize a member variable processors_dict for you which will contain initialization of all of the processors you specified and will need to process your complete batch.

Rest it behaves in same way, expects an item and returns an item which can be of any type.

class mmf.datasets.processors.processors.**BatchProcessorConfigType**(*processors: mmf.common.typings.ProcessorConf*

class mmf.datasets.processors.processors.**CaptionProcessor**(*config*, **args*, ***kwargs*)

Processes a caption with start, end and pad tokens and returns raw string.

Parameters **config** (*DictConfig*) – Configuration for caption processor.

class mmf.datasets.processors.processors.**CopyProcessor**(*config*, **args*, ***kwargs*)

Copy boxes from numpy array

class mmf.datasets.processors.processors.**DETRImageAndTargetProcessor**(*config*, **args*, ***kwargs*)

Process a detection image and target in consistent with DETR. At training time, random crop is done. At test time, an image is deterministically resized with short side equal to *image_size* (while ensuring its long side no larger than *max_size*)

class mmf.datasets.processors.processors.**EvalAIAnswerProcessor**(**args*, ***kwargs*)

Processes an answer similar to Eval AI

class mmf.datasets.processors.processors.**FastTextProcessor**(*config*, **args*, ***kwargs*)

FastText processor, similar to GloVe processor but returns FastText vectors.

Parameters **config** (*DictConfig*) – Configuration values for the processor.

class mmf.datasets.processors.processors.**GloVeProcessor**(*config*, **args*, ***kwargs*)

Inherits VocabProcessor, and returns GloVe vectors for each of the words. Maps them to index using vocab processor, and then gets GloVe vectors corresponding to those indices.

Parameters **config** (*DictConfig*) – Configuration parameters for GloVe same as *VocabProcessor()*.

class mmf.datasets.processors.processors.**GraphVQAAnswerProcessor**(*config*, **args*, ***kwargs*)

Processor for generating answer scores for answers passed using VQA accuracy formula. Using VocabDict class to represent answer vocabulary, so parameters must specify "vocab_file". "num_answers" in parameter config specify the max number of answers possible. Takes in dict containing "answers" or "answers_tokens". "answers" are preprocessed to generate "answers_tokens" if passed.

This version also takes a graph vocab and predicts a main and graph stream simultanously

Parameters **config** (*DictConfig*) – Configuration for the processor

answer_vocab

Class representing answer vocabulary

Type VocabDict

compute_answers_scores(*answers_indices*)

Generate VQA based answer scores for answers_indices.

Parameters **answers_indices** (*torch.LongTensor*) – tensor containing indices of the answers

Returns tensor containing scores.

Return type torch.FloatTensor

get_true_vocab_size()

True vocab size can be different from normal vocab size in some cases such as soft copy where dynamic answer space is added.

Returns True vocab size.

Return type int

get_vocab_size()
Get vocab size of the answer vocabulary. Can also include soft copy dynamic answer space size.

Returns size of the answer vocabulary

Return type int

idx2word(*idx*)
Index to word according to the vocabulary.

Parameters **idx** (*int*) – Index to be converted to the word.

Returns Word corresponding to the index.

Return type str

word2idx(*word*)
Convert a word to its index according to vocabulary

Parameters **word** (*str*) – Word to be converted to index.

Returns Index of the word.

Return type int

class mmf.datasets.processors.processors.**M4CAnswerProcessor**(*config*, **args*, ***kwargs*)
Process a TextVQA answer for iterative decoding in M4C

match_answer_to_vocab_ocr_seq(*answer*, *vocab2idx_dict*, *ocr2inds_dict*, *max_match_num=20*)
Match an answer to a list of sequences of indices each index corresponds to either a fixed vocabulary or an OCR token (in the index address space, the OCR tokens are after the fixed vocab)

class mmf.datasets.processors.processors.**M4CCaptionProcessor**(*config*, **args*, ***kwargs*)

class mmf.datasets.processors.processors.**MaskedRegionProcessor**(*config*, **args*, ***kwargs*)
Masks a region with probability *mask_probability*

class mmf.datasets.processors.processors.**MultiClassFromFile**(*config: mmf.datasets.processors.processors.MultiCla*
**args*, ***kwargs*)
Label processor for multi class cases where the labels are saved in a file.

class mmf.datasets.processors.processors.**MultiClassFromFileConfig**(*vocab_file: str*)

class mmf.datasets.processors.processors.**MultiHotAnswerFromVocabProcessor**(*config*, **args*, ***kwargs*)

compute_answers_scores(*answers_indices*)
Generate VQA based answer scores for answers_indices.

Parameters **answers_indices** (*torch.LongTensor*) – tensor containing indices of the answers

Returns tensor containing scores.

Return type torch.FloatTensor

class mmf.datasets.processors.processors.**PhocProcessor**(*config*, **args*, ***kwargs*)
Compute PHOC features from text tokens

class mmf.datasets.processors.processors.**Processor**(*config: mmf.common.typings.ProcessorConfigType*, **args*, ***kwargs*)
Wrapper class used by MMF to initialized processor based on their `type` as passed in configuration. It retrieves the processor class registered in registry corresponding to the `type` key and initializes with `params` passed in configuration. All functions and attributes of the processor initialized are directly available via this class.

Parameters **config** (*DictConfig*) – DictConfig containing `type` of the processor to be initialized and `params` of that processor.

class mmf.datasets.processors.processors.**SimpleSentenceProcessor**(**args*, ***kwargs*)
Tokenizes a sentence and processes it.

tokenizer
Type of tokenizer to be used.

Type function

class mmf.datasets.processors.processors.**SimpleWordProcessor**(**args*, ***kwargs*)
Tokenizes a word and processes it.

tokenizer
Type of tokenizer to be used.

Type function

class mmf.datasets.processors.processors.**SoftCopyAnswerProcessor**(*config*, **args*, ***kwargs*)
Similar to Answer Processor but adds soft copy dynamic answer space to it. Read https://arxiv.org/abs/1904.08920 for extra information on soft copy and LoRRA.

Parameters **config** (*DictConfig*) – Configuration for soft copy processor.

get_true_vocab_size()
Actual vocab size which only include size of the vocabulary file.

Returns Actual size of vocabs.

Return type int

get_vocab_size()
Size of Vocab + Size of Dynamic soft-copy based answer space

Returns Size of vocab + size of dynamic soft-copy answer space.

Return type int

class mmf.datasets.processors.processors.**TransformerBboxProcessor**(*config*, **args*, ***kwargs*)
Process a bounding box and returns a array of normalized bbox positions and area

class mmf.datasets.processors.processors.**VQAAnswerProcessor**(*config*, **args*, ***kwargs*)
Processor for generating answer scores for answers passed using VQA accuracy formula. Using VocabDict class to represent answer vocabulary, so parameters must specify "vocab_file". "num_answers" in parameter config specify the max number of answers possible. Takes in dict containing "answers" or "answers_tokens". "answers" are preprocessed to generate "answers_tokens" if passed.

Parameters **config** (*DictConfig*) – Configuration for the processor

answer_vocab
Class representing answer vocabulary

Type VocabDict

compute_answers_scores(*answers_indices*)
Generate VQA based answer scores for answers_indices.

Parameters **answers_indices** (*torch.LongTensor*) – tensor containing indices of the answers

Returns tensor containing scores.

Return type torch.FloatTensor

get_true_vocab_size()
True vocab size can be different from normal vocab size in some cases such as soft copy where dynamic answer space is added.

Returns True vocab size.

Return type int

get_vocab_size()
Get vocab size of the answer vocabulary. Can also include soft copy dynamic answer space size.

Returns size of the answer vocabulary

Return type int

idx2word(*idx*)
Index to word according to the vocabulary.

Parameters **idx** (*int*) – Index to be converted to the word.

Returns Word corresponding to the index.

Return type str

word2idx(*word*)
Convert a word to its index according to vocabulary

Parameters **word** (*str*) – Word to be converted to index.

Returns Index of the word.

Return type int

class mmf.datasets.processors.processors.**VocabProcessor**(*config*, **args*, ***kwargs*)
Use VocabProcessor when you have vocab file and you want to process words to indices. Expects UNK token as "<unk>" and pads sentences using "<pad>" token. Config parameters can have `preprocessor` property which is used to preprocess the item passed and `max_length` property which points to maximum length of the sentence/tokens which can be convert to indices. If the length is smaller, the sentence will be padded. Parameters for "vocab" are necessary to be passed.

Key: vocab

Example Config:

```
dataset_config:
  vqa2:
    data_dir: ${env.data_dir}
    processors:
      text_processor:
        type: vocab
```

(continues on next page)

(continued from previous page)

```
        params:
          max_length: 14
          vocab:
            type: intersected
            embedding_name: glove.6B.300d
            vocab_file: vqa2/defaults/extras/vocabs/vocabulary_100k.txt
```

Parameters **config** (*DictConfig*) – node containing configuration parameters of the processor

vocab
Vocab class object which is abstraction over the vocab file passed.

Type Vocab

get_pad_index()
Get index of padding <pad> token in vocabulary.

Returns index of the padding token.

Return type int

get_vocab_size()
Get size of the vocabulary.

Returns size of the vocabulary.

Return type int

class mmf.datasets.processors.image_processors.**GrayScaleTo3Channels**(**args*, ***kwargs*)

class mmf.datasets.processors.image_processors.**NormalizeBGR255**(**args*, ***kwargs*)

class mmf.datasets.processors.image_processors.**ResizeShortest**(**args*, ***kwargs*)

class mmf.datasets.processors.image_processors.**TorchvisionTransforms**(*config*, **args*, ***kwargs*)

class mmf.datasets.processors.bert_processors.**BertTokenizer**(*config*, **args*, ***kwargs*)

class mmf.datasets.processors.bert_processors.**MaskedRobertaTokenizer**(*config*, **args*, ***kwargs*)

_convert_to_indices(*tokens_a: List[str], tokens_b: Optional[List[str]] = None, probability: float = 0.15*) → Dict[str, torch.Tensor]
Roberta encodes - single sequence: `<s> X </s>` - pair of sequences: `<s> A </s> </s> B </s>`

_truncate_seq_pair(*tokens_a: List[str], tokens_b: List[str], max_length: int*)
Truncates a sequence pair in place to the maximum length.

class mmf.datasets.processors.bert_processors.**MaskedTokenProcessor**(*config*, **args*, ***kwargs*)

_convert_to_indices(*tokens_a: List[str], tokens_b: Optional[List[str]] = None, probability: float = 0.15*) → Dict[str, torch.Tensor]
BERT encodes - single sequence: `[CLS] X [SEP]` - pair of sequences: `[CLS] A [SEP] B [SEP]`

_truncate_seq_pair(*tokens_a: List[str], tokens_b: List[str], max_length: int*)
Truncates a sequence pair in place to the maximum length.

class mmf.datasets.processors.bert_processors.**MultiSentenceBertTokenizer**(*config*, **args*, ***kwargs*)
Extension of BertTokenizer which supports multiple sentences. Separate from normal usecase, each sentence will be passed through bert tokenizer separately and indices will be reshaped as single tensor. Segment ids will also be increasing in number.

class mmf.datasets.processors.bert_processors.**MultiSentenceRobertaTokenizer**(*config*, **args*, ***kwargs*)
Extension of SPMTokenizer which supports multiple sentences. Similar to MultiSentenceBertTokenizer.

class mmf.datasets.processors.bert_processors.**RobertaTokenizer**(*config*, **args*, ***kwargs*)

1.9 utils.text

Text utils module contains implementations for various decoding strategies like Greedy, Beam Search and Nucleus Sampling.

In your model's config you can specify `inference` attribute to use these strategies in the following way:

```
model_config:
    some_model:
        inference:
            - type: greedy
            - params: {}
```

class mmf.utils.text.**BeamSearch**(*vocab*, *config*)

class mmf.utils.text.**NucleusSampling**(*vocab*, *config*)
Nucleus Sampling is a new text decoding strategy that avoids likelihood maximization. Rather, it works by sampling from the smallest set of top tokens which have a cumulative probability greater than a specified threshold.

Present text decoding strategies like beam search do not work well on open-ended generation tasks (even on strong language models like GPT-2). They tend to repeat text a lot and the main reason behind it is that they try to maximize likelihood, which is a contrast from human-generated text which has a mix of high and low probability tokens.

Nucleus Sampling is a stochastic approach and resolves this issue. Moreover, it improves upon other stochastic methods like top-k sampling by choosing the right amount of tokens to sample from. The overall result is better text generation on the same language model.

Link to the paper introducing Nucleus Sampling (Section 6) - https://arxiv.org/pdf/1904.09751.pdf

Parameters

- **vocab** (*list*) – Collection of all words in vocabulary.
- **sum_threshold** (*float*) – Ceiling of sum of probabilities of tokens to sample from.

class mmf.utils.text.**TextDecoder**(*vocab*)
Base class to be inherited by all decoding strategies. Contains implementations that are common for all strategies.

Parameters **vocab** (*list*) – Collection of all words in vocabulary.

mmf.utils.text.**generate_ngrams**(*tokens*, *n=1*)
Generate ngrams for particular 'n' from a list of tokens

Parameters

- **tokens** (*List[str]*) – List of tokens for which the ngram are to be generated
- **n** (*int, optional*) – n for which ngrams are to be generated. Defaults to 1.

Returns List of ngrams generated.

Return type List[str]

mmf.utils.text.**generate_ngrams_range**(*tokens*, *ngram_range=(1, 3)*)
Generates and returns a list of ngrams for all n present in ngram_range

Parameters

- **tokens** (*List[str]*) – List of string tokens for which ngram are to be generated
- **ngram_range** (*List[int], optional*) – List of 'n' for which ngrams are to be generated. For e.g. if ngram_range = (1, 4) then it will returns 1grams, 2grams and 3grams. Defaults to (1, 3).

Returns List of ngrams for each n in ngram_range

Return type List[str]

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

m

Index

Symbols

A

B

C

D

E

F

G

I

L

M

N

O

P

R

S

T

U

V

W