MMF Documentation

Release 1.0.0rc8

Facebook AI Research

Jun 02, 2020

Getting Started

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.

CHAPTER 1

Citation

If you use MMF in your work, please cite:

```
@inproceedings{singh2019pythia,
    title={Pythia-a platform for vision \& language research},
    author={Singh, Amanpreet and Natarajan, Vivek and Jiang, Yu and Chen, Xinlei and␣
↪Shah, Meet and Rohrbach, Marcus and Batra, Dhruv and Parikh, Devi},
    booktitle={SysML Workshop, NeurIPS},
    volume={2018},
    year={2019}
}
```

1.1 Installation

MMF is tested on Python 3.6+ and PyTorch 1.5.

1.1.1 Install using pip

MMF can be installed from pip with following command:

```
pip install --upgrade --pre mmf
```

Use this if:

- You are using MMF as a library and not developing inside MMF. Have a look at extending MMF tutorial.
- You want easy installation and don't care about up-to-date features. Note that, pip packages are always outdated compared to installing from source

1.1.2 Install from source

To install from source, do:

```
git clone https://github.com/facebookresearch/mmf.git
cd mmf
pip install --editable .
```

1.1.3 Running tests

MMF uses pytest for testing purposes. To ensure everything and run tests at your end do:

```
pytest ./tests/
```

1.2 Quickstart

Authors: Amanpreet Singh

In this quickstart, we are going to train M4C model on TextVQA. Follow instructions at the bottom to train other models in MMF.

1.2.1 Installation

Install MMF following the installation documentation.

1.2.2 Getting Data

In MMF datasets and required files will be downloaded automatically when we run training next. For more details about custom datasets and other advanced setups for datasets check the *dataset documentation*.

1.2.3 Training

Now we can start training by running the following command:

```
mmf_run config=projects/m4c/configs/textvqa/defaults.yaml datasets=textvqa model=m4c␣
↪run_type=train_val
```

1.2.4 Inference

For running inference or generating predictions, we can specify a pretrained model using its zoo key and then run the following command:

```
mmf_predict config=projects/m4c/configs/textvqa/defaults.yaml datasets=textvqa␣
↪model=m4c run_type=test checkpoint.resume_zoo=m4c.textvqa.defaults
```

For running inference on `val` set, use `run_type=val` and rest of the arguments remain same. Check more details in pretrained models section.

These commands should be enough to get you started with training and performing inference using MMF.

1.2.5 Citation

If you use MMF in your work, please cite:

```
@inproceedings{singh2019pythia,
  title={Pythia-a platform for vision \& language research},
  author={Singh, Amanpreet and Natarajan, Vivek and Jiang, Yu and Chen, Xinlei and␣
↪Shah, Meet and Rohrbach, Marcus and Batra, Dhruv and Parikh, Devi},
  booktitle={SysML Workshop, NeurIPS},
  volume={2018},
  year={2019}
}
```

1.2.6 Next steps

To dive deep into world of MMF, you can move on the following next topics:

- *Concepts and Terminology*
- Using Pretrained Models
- Challenge Participation
- FAQs

1.3 Features

MMF features:

- **Model Zoo**: Reference implementations for state-of-the-art vision and language model including LoRRA (SoTA on VQA and TextVQA), Pythia model (VQA 2018 challenge winner), BAN and BUTD.
- **Multi-Tasking**: Support for multi-tasking which allows training on multiple datasets together.
- **Datasets**: Includes support for various datasets built-in including VQA, VizWiz, TextVQA, VisualDialog, MS COCO Captioning.
- **Modules**: Provides implementations for many commonly used layers in vision and language domain
- **Distributed**: Support for distributed training based on DataParallel as well as DistributedDataParallel.
- **Unopinionated**: Unopinionated about the dataset and model implementations built on top of it.
- **Customization**: Custom losses, metrics, scheduling, optimizers, tensorboard; suits all your custom needs.

You can use MMF to **bootstrap** for your next vision and language multimodal research project.

MMF can also act as **starter codebase** for challenges around vision and language datasets (TextVQA challenge, VQA challenge).

1.4 Pretrained Models

[Outdated] A new version of this will be uploaded soon

Performing inference using pretrained models in MMF is easy. Pickup a pretrained model from the table below and follow the steps to do inference or generate predictions for EvalAI evaluation. This section expects that you have already installed the required data as explained in quickstart.

Model	Model Key	Supported Datasets	Pretrained Models	Notes
Pythia	pythia	vqa2, vizwiz, textvqa, visual_genome,	vqa2 train+val, vqa2 train only, vizwiz	VizWiz model has been pretrained on VQAv2 and transferred
LoRRA	lorra	vqa2, vizwiz, textvqa	textvqa	
CNNL-STM	cnn_lstm	clevr		Features are calculated on fly in this on
BAN	ban	vqa2, vizwiz, textvqa	Coming soon!	Support is preliminary and haven't been tested throughly.
BUTD	butd	coco	coco	

Now, let's say your link to pretrained model `model` is `link` (select from table > right click > copy link address), the respective config should be at `configs/[task]/[dataset]/[model].yaml`. For example, config file for `vqa2 train_and_val` should be `configs/vqa/vqa2/pythia_train_and_val.yaml`. Now to run inference for EvalAI, run the following command.

```
cd ~/mmf/data
mkdir -p models && cd models;
# Download the pretrained model
wget [link]
cd ../..;
python tools/run.py --datasets [dataset] --model [model] --config [config] \
--run_type inference --evalai_inference 1 --resume_file data/[model].pth
```

If you want to train or evaluate on val, change the `run_type` to `train` or `val` accordingly. You can also use multiple run types, for e.g. to do training, inference on val as well as test you can set `--run_type` to `train+val+inference`.

if you remove `--evalai_inference` argument, Pythia will perform inference and provide results directly on the dataset. Do note that this is not possible in case of test sets as we don't have answers/targets for them. So, this can be useful for performing inference on val set locally.

Table below shows evaluation metrics for various pretrained models:

Model	Dataset	Metric	Notes
Pythia	vqa2 (train+val)	test-dev accuracy - 68.31%	Can be easily pushed to 69.2%
Pythia	vqa2 (train)	test-dev accuracy - 66.7%	
Pythia	vizwiz (train)	test-dev accuracy - 54.22%	Pretrained on VQA2 and transferred to VizWiz
LoRRA	textvqa (train)	val accuracy - 27.4%	
BUTD	coco (karpathy-train)	BLEU 1 - 76.02, BLEU 4- 35.42 METEOR- 27.39, ROUGE_L- 56.17 CIDEr - 112.03, SPICE - 20.33	Beam Search(length 5), Karpathy test split

Note for BUTD model : For training BUTD model use the config `butd.yaml`. Training uses greedy decoding for validation. Currently we do not have support to train the model using beam search decoding validation. We will add that support soon. For inference only use `butd_beam_search.yaml` config that supports beam search decoding.

Note that, for simplicity, our current released model **does not** incorporate extensive data augmentations (e.g. visual genome, visual dialogue) during training, which was used in our challenge winner entries for VQA and VizWiz 2018. As a result, there can be some performance gap to models reported and released previously. If you are looking for reproducing those results, please checkout the v0.1 release.

1.5 Configuration System

MMF relies on OmegaConf for its configuration system and adds some sugar on top of it. We have developed MMF as a config-first framework. Most of the parameters/settings in MMF are configurable. MMF defines some default configuration settings for its system including datasets and models. Users can then update these values via their own config or a command line dotlist.

TL;DR

- MMF uses OmegaConf for its configuration system with some sugar on top.
- MMF defines *base defaults config* containing all MMF specific parameters and then each dataset and model define their own configs (example configs: *[model] [dataset]*).
- The user can define its own config specified by `config=<x>` at command line for each unique experiment or training setup. This has higher priority then base, model and dataset default configs and can override anything in those.
- Finally, user can override (highest priority) the final config generated by merge of all above configs by specifying config parameters as dotlist in their command. This is the **recommended** way of overriding the config parameters in MMF.
- How MMF knows which config to pick for dataset and model? The user needs to specify those in his command as `model=x` and `dataset=y`.
- Some of the MMF config parameters under `env` field can be overridden by environment variable. Have a look at them.

1.5.1 OmegaConf

For understanding and using the MMF configuration system to its full extent having a look at OmegaConf docs especially the sections on interpolation, access and configuration flags. MMF's config currently is by default in struct mode and we plan to make it readonly in future.

1.5.2 Hierarchy

MMF follows set hierarchy rules to determine the final configuration values. Following list shows the building blocks of MMF's configuration in an increasing order of priority (higher rank will override lower rank).

- *Base Defaults Config*
- Dataset's Config (defined in dataset's `config_path` classmethod)
- Model's Config (defined in model's `config_path` classmethod)
- User's Config (Passed by user as `config=x` in command)
- Command Line DotList (Passed by user as `x.y.z=v` dotlist in command)

Note: Configs other than base defaults can still add new nodes that are not in base defaults config, so user can add their own config parameters if they need to without changing the base defaults. If a node has same path, nodes in higher priority config will override the lower priority nodes.

1.5.3 Base Defaults

Full base defaults config can be seen *below*. This config is base of MMF's configuration system and is included in all of the experiments. It sets up nodes for training related configuration and those that need to be filled by other configs which are specified by user. Main configuration parameters that base defaults define:

- training parameters
- distributed training parameters
- env parameters
- evaluation parameters
- checkpoint parameters
- run_type parameters

1.5.4 Dataset Config

Each dataset *registered* to MMF can define its defaults config by specifying it in classmethod `config_path` (example). If `processors` key whose value is a dictionary is specified, processors will be initialized by the dataset builder. If dataset builder inherits from MMFDatasetBuilder, it will look for `annotations`, `features` and `images` field as well in the configuration. A sample config for a builder inheriting MMFDatasetBuilder would look like:

```
dataset_config:
    dataset_registry_key:
        use_images: true
        use_features: true
        annotations:
            train:
            - ...
            val:
            - ...
            test:
            - ...
        images:
            train:
            - ...
            val:
            - ...
            test:
            - ...
        features:
            train:
            - ...
            val:
            - ...
            test:
            - ...
        processors:
```

(continues on next page)

(continued from previous page)

```
            text_processor:
                type: x
                params: ...
```

Configs for datasets packages with MMF are present at mmf/configs/datasets. Each dataset also provides composable configs which can be used to use some different from default but standard variation of the datasets. These can be directly included into user config by using *includes* directive.

User needs to specify the dataset they are using by adding `dataset=<dataset_key>` option to their command.

1.5.5 Model Config

Similar to dataset config, each model *registered* to MMF can define its config. this is defined by model's `config_path` classmethod (example). Configs for models live at mmf/configs/models. Again, like datasets models also provide some variations which can be used by including configs for those variations in the user config.

User needs to specify the model they want to use by adding `model=<model_key>` option to their command. A sample model config would look like:

```
model_config:
    model_key:
        random_module: ...
```

1.5.6 User Config

User can specify their configuration specific to an experiment or training setup by adding `config=<config_path>` argument to their command. User config can specify for e.g. training parameters according to their experiment such as batch size using `training.batch_size`. Most common use case for user config is to specify optimizer, scheduler and training parameters. Other than that user config can also include configs for variations of models and datasets they want to test on. Have a look at an example user config *here*.

1.5.7 Command Line Dot List Override

Updating the configuration through dot list syntax is very helpful when running multiple versions of an experiment without actually updating a config. For example, to override batch size from command line you can add `training.batch_size=x` at the end of your command. Similarly, for overriding an annotation in the hateful memes dataset, you can do `dataset_config.hateful_memes.annotations.train[0]=x`.

Note: Command Line Dot List overrides are our recommended way of updating config parameters instead of manually updating them in config for every other change.

1.5.8 Includes

MMF's configuration system on top of OmegaConf allows building user configs by including composable configs provided by the datasets and models. You can include it following the syntax

```
includes:
- path/to/first/yaml/to/be/included.yaml
- second.yaml
```

The configs will override in the sequence of how they appear in the directive. Finally, the config parameters defined in the current config will override what is present in the includes. So, for e.g.

First file, `a.yaml`:

```yaml
# a.yaml
dataset_config:
  hateful_memes:
    max_features: 80
    use_features: true
  vqa2:
    use_features: true

model_config:
  mmbt:
    num_classes: 4
    features_dim: 2048
```

Second file, `b.yaml`:

```yaml
# b.yaml
optimizer:
  type: adam

dataset_config:
  hateful_memes:
    max_features: 90
    use_features: false
    use_images: true
  vqa2:
    depth_first: false
```

And final user config, `user.yaml`:

```yaml
# user.yaml
includes:
- a.yaml
- b.yaml

dataset_config:
  hateful_memes:
    max_features: 100
  vqa2:
    annotations:
      train: x.npy

model_config:
  mmbt:
    num_classes: 2
```

would result in final config:

```yaml
dataset_config:
  hateful_memes:
    max_features: 100
    use_features: false
    use_images: true
  vqa2:
```

(continues on next page)

(continued from previous page)

```
    use_features: true
    depth_first: false
    annotations:
      train: x.npy

model_config:
  mmbt:
    num_classes: 2
    features_dim: 2048

optimizer:
  type: adam
```

1.5.9 Other overrides

We also support some useful overrides schemes at the same level of command line dot list override. For example, user can specify their overrides in form of demjson as value to argument `--config_override` which will them override each part of config accordingly.

1.5.10 Environment Variables

MMF supports overriding some of the config parameters through environment variables. Have a look at them in *base default config*'s `env` parameters.

1.5.11 Base Defaults Config

Have a look at the defaults config of MMF along with description of parameters from which you may need to override parameters for your experiments:

```
# Configuration version is useful in migrating older configs to new ones
config_version: 1.0

# Configuration for training
training:
    # Name of the trainer class used to define the training/evalution loop
    trainer: base_trainer
    # Seed to be used for training. -1 means random seed between 1 and 100000.
    # Either pass fixed through your config or command line arguments
    # Pass null to the seed if you don't want it seeded anyhow and
    # want to leave it to default
    seed: -1
    # Name of the experiment, will be used while saving checkpoints
    # and generating reports
    experiment_name: run
    # Maximum number of iterations the training will run
    max_updates: 22000
    # Maximum epochs in case you don't want to use max_updates
    # Can be mixed with max iterations, so it will stop whichever is
    # completed first. Default: null means epochs won't be used
    max_epochs: null

    # After `log_interval` iterations, current iteration's training loss will be
```

(continues on next page)

(continued from previous page)

```
    # reported. This will also report validation on a single batch from validation set
    # to provide an estimate on validation side
    log_interval: 100
    # Level of logging, only logs which are >= to current level will be logged
    logger_level: info
    # Log format: json, simple
    log_format: simple
    # Whether to log detailed final configuration parameters
    log_detailed_config: false
    # Whether MMF should log or not, Default: False, which means
    # mmf will log by default
    should_not_log: false

    # Tensorboard control, by default tensorboard is disabled
    tensorboard: false

    # Size of each batch. If distributed or data_parallel
    # is used, this will be divided equally among GPUs
    batch_size: 512
    # Number of workers to be used in dataloaders
    num_workers: 4
    # Some datasets allow fast reading by loading everything in the memory
    # Use this to enable it
    fast_read: false
    # Use in multi-tasking, when you want to sample tasks proportional to their sizes
    dataset_size_proportional_sampling: true
    # Whether to pin memory in dataloader
    pin_memory: false

    # After `checkpoint_interval` iterations, MMF will make a snapshot
    # which will involve creating a checkpoint for current training scenarios
    checkpoint_interval: 1000
    # This will evaluate evaluation metrics on whole validation set after
    # evaluation interval
    evaluation_interval: 1000
    # Whether gradients should be clipped
    clip_gradients: false
    # Mode for clip norm
    clip_norm_mode: all

    early_stop:
        # Whether to use early stopping, (Default: false)
        enabled: false
        # Patience for early stoppings
        patience: 4000
        # Criteria to be monitored for early stopping
        # total_loss will monitor combined loss from all of the tasks
        # Criteria can also be an evaluation metric in this format `dataset/metric`
        # for e.g. vqa2/vqa_accuracy
        criteria: total_loss
        # Whether the monitored criteria should be minimized for early stopping
        # or not, for e.g. you would want to minimize loss but maximize an evaluation
        # metric like accuracy etc.
        minimize: true

    # Should a lr scheduler be used
    lr_scheduler: false
```

(continues on next page)

(continued from previous page)

```yaml
    # DEPRECATED: Look at scheduler_attributes or
    # Use PythiaScheduler directly instead
    # Steps for LR scheduler, will be an array of iteration count
    # when lr should be decreased
    lr_steps: []
    # DEPRECATED: Look at scheduler_attributes or
    # Use PythiaScheduler directly instead
    # Ratio for each lr step
    lr_ratio: 0.1

    # NOTE: Have a look at newer scheduler available in MMF (such as AdamW) before
    # using these options
    # Should use warmup for lr
    use_warmup: false
    # Warmup factor learning rate warmup
    warmup_factor: 0.2
    # Iteration until which warnup should be done
    warmup_iterations: 1000

    # Device on which the model will be trained. Set 'cpu' to train/infer on CPU
    device: cuda
    # Local rank of the GPU device
    local_rank: null

    # If verbose dump is active, MMF will dump dataset, model specific
    # information which can be useful in debugging
    verbose_dump: false

    # Turn on if you want to ignore unused parameters in case of DDP
    find_unused_parameters: false

    # By default metrics evaluation is turned off during training. Set this to true
    # to enable evaluation every log_interval
    evaluate_metrics: false

# Configuration for evaluation
evaluation:
    # Metrics for evaluation
    metrics: []
    # Generate predictions in a file
    predict: false
    # Prediction file format (csv|json), default is json
    predict_file_format: json

# Configuration for models, default configuration files for various models
# included in MMF can be found under configs directory in root folder
model_config: {}

# Configuration for datasets. Separate configuration
# for different datasets included in MMF are included in dataset folder
# which can be mixed and matched to train multiple datasets together
# An example for mixing all vqa datasets is present under vqa folder
dataset_config: {}

# Defines which datasets from the above tasks you want to train on
datasets: []
```

(continues on next page)

(continued from previous page)

```yaml
# Defines which model you want to train on
model: null

# Config file to be optionally passed by the user
config: null

# Type of run, train+inference by default means both training and inference
# (test) stage will be run, if run_type contains 'val',
# inference will be run on val set also.
run_type: train_inference

# Configuration for optimizer, examples can be found in models' configs in
# configs folder
optimizer: {}

# Configuration for scheduler, examples can be found in models' configs
scheduler: {}

# Common environment configurations for MMF
env:
    # Universal cache directory for mmf
    # This can be overridden by using MMF_CACHE_DIR environment variable
    # or by directly setting this configuration attribute env.cache_dir
    # If nothing is specified, default is set to "mmf" inside
    # pytorch's cache folder
    cache_dir: ${resolve_cache_dir:MMF_CACHE_DIR}

    # Config path for dataset zoo, can be overridden via environment
    # variable MMF_DATASET_ZOO as well.
    dataset_zoo: ${env:MMF_DATASET_ZOO,configs/zoo/datasets.yaml}
    model_zoo: ${env:MMF_MODEL_ZOO, configs/zoo/models.yaml}

    # Similar to cache dir, but can be used if specifically want to override
    # where MMF stores your data. Default would be cache_dir/data.
    # We will auto download models and datasets in this folder
    data_dir: ${resolve_dir:MMF_DATA_DIR, data}

    # Directory for saving checkpoints and other metadata
    # Use MMF_SAVE_DIR or env.save_dir to override
    save_dir: ${env:MMF_SAVE_DIR, ./save}

    # Directory for saving logs, default is "logs" inside the save folder
    # If log_dir is specifically passed, logs will be written inside that folder
    # Use MMF_LOG_DIR or env.log_dir to override
    log_dir: ${env:MMF_LOG_DIR,}

    # Directory for saving reports, if not passed a opts based folder will be␣
↪generated
    # inside save_dir/reports and reports will be saved there
    # Use MMF_REPORT_DIR or env.report_dir to override
    report_dir: ${env:MMF_REPORT_DIR,}

    # Log directory for tensorboard, default points to same as logs
    # Only used when training.tensorboard is enabled.
    # Use MMF_TENSORBOARD_LOGDIR or env.tensorboard_logdir to override
    tensorboard_logdir: ${env:MMF_TENSORBOARD_LOGDIR,}
```

(continues on next page)

(continued from previous page)

```
    # User directory where user can keep their own models independent of MMF
    # This allows users to create projects which only include MMF as dependency
    # Use MMF_USER_DIR or env.user_dir to specify
    user_dir: ${env:MMF_USER_DIR,}

###
# Configuration for the distributed setup
distributed:
    ###
    # Typically tcp://hostname:port that will be used to establish initial connection
    init_method: null
    # Rank of the current worker
    rank: 0
    # Port number, not required if using init_method,
    port: -1
    # Backend for distributed setup
    backend: nccl
    # Total number of GPUs across all nodes (default: all visible GPUs)
    world_size: ${device_count:}
    # Set if you do not want spawn multiple processes even if
    # multiple GPUs are visible
    no_spawn: false

# Configuration for checkpointing including resuming and loading pretrained models
checkpoint:
    # If checkpoint.resume is true, MMF will try to load automatically load
    # checkpoint and state from "current.ckpt" from env.save_dir
    resume: false
    # `checkpoint.resume_file` can be used to load a specific checkpoint from a file
    # Can also be a zoo key
    resume_file: null
    # `checkpoint.resume_best` will load the best checkpoint according to
    # training.early_stop.criteria instead of the last saved ckpt
    resume_best: false
    # `checkpoint.resume_pretrained` can be used in conjuction with `resume_file`
    # or `resume_zoo` where you specify a checkpoint or .pth file to be loaded
    # but it is mapped based on `checkpoint.pretrained_state_mapping`
    # For e.g. if you want to resume from visual_bert pretrained on coco
    # You would set `checkpoint.resume_zoo=visual_bert.pretrained.coco` and
    # then set `checkpoint.resume_pretrained=True` which will then pick up
    # only the base which you would define in the `checkpoint.pretrained_state_
↪mapping`
    resume_pretrained: false
    # `checkpoint.resume_zoo` can be used to resume from a pretrained model provided
    # in zoo. Value maps to key from zoo. `checkpoint.resume_file` has higher
    # priority compared to `checkpoint.resume_zoo`.
    resume_zoo: null
    # `checkpoint.zoo_config_override` will override the current model config of␣
↪trainer
    # with what is provided from the zoo checkpoint and will load the model
    # using .from_pretrained of the model passed
    zoo_config_override: false
    # `checkpoint.pretrained_state_mapping` specifies how exactly a pretrained
    # model will be loaded and mapped to which keys of the target model
    # Only use if the keys on the model in which pretrained model is to be loaded
    # don't match with those of the pretrained model or you only want to load specific
```

(continues on next page)

(continued from previous page)

```
    # item from the pretrained model. `checkpoint.resume_pretrained` must be
    # true to use this mapping. for e.g. you can specify
    # text_embedding: text_embedding_pythia
    # for loading `text_embedding` module of your model from `text_embedding_pythia`of
    # pretrained file specified in `checkpoint.resume_file`.
    pretrained_state_mapping: {}

    # Whether to save git details or not
    save_git_details: true

    # `checkpoint.reset` configuration defines what exactly should be reset
    # in case the file from which we are resuming is .ckpt and not .pth
    reset:
        # Everything will be reset except the state_dict of model
        all: false
        # Optimizer specifically will be reset
        optimizer: false
        # All counts such as best_update, current_iteration etc will be reset
        counts: false
```

1.6 Challenge Participation

[Outdated] A new version of this will be uploaded soon

Participating in EvalAI challenges is really easy using MMF. We will show how to do inference for two challenges here:

Note: This section assumes that you have downloaded data following the Quickstart tutorial.

1.6.1 TextVQA challenge

TextVQA challenge is available at this link. Currently, LoRRA is the SoTA on TextVQA. To do inference on val set using LoRRA, follow the steps below:

```
# Download the model first
cd ~/mmf/data
mkdir -p models && cd models;
# Get link from the table above and extract if needed
wget https://dl.fbaipublicfiles.com/pythia/pretrained_models/textvqa/lorra_best.pth

cd ../..
# Replace datasets and model with corresponding key for other pretrained models
python tools/run.py --datasets textvqa --model lorra --config configs/vqa/textvqa/
↪lorra.yaml \
--run_type val --evalai_inference 1 --resume_file data/models/lorra_best.pth
```

In the printed log, MMF will mention where it wrote the JSON file it created. Upload that file on EvalAI:

```
> Go to https://evalai.cloudcv.org/web/challenges/challenge-page/244/overview
> Select Submit Tab
> Select Validation Phase
```

(continues on next page)

(continued from previous page)

```
> Select the file by click Upload file
> Write a model name
> Upload
```

To check your results, go in 'My submissions' section and select 'Validation Phase' and click on 'Result file'.

Now, you can either edit the LoRRA model to create your own model on top of it or create your own model inside MMF to beat LoRRA in challenge.

1.6.2 VQA Challenge

Similar to TextVQA challenge, VQA Challenge is available at this link. You can either select Pythia as your base model or LoRRA model (available soon for VQA2) from the table in pretrained models section as a base.

Follow the same steps above, replacing `--model` with `pythia` or `lorra` and `--datasets` with `vqa2`. Also, replace the config accordingly. Here are example commands for using Pythia to do inference on test set of VQA2.

```
# Download the model first
cd ~/mmf/data
mkdir -p models && cd models;
# Get link from the table above and extract if needed
wget https://dl.fbaipublicfiles.com/pythia/pretrained_models/textvqa/pythia_train_val.
↪pth

cd ../..
# Replace datasets and model with corresponding key for other pretrained models
python tools/run.py --datasets vqa2 --model pythia --config configs/vqa/vqa2/pythia.
↪yaml \
--run_type inference --evalai_inference 1 --resume_file data/models/pythia_train_val.
↪pth
```

Now, similar to TextVQA challenge follow the steps to upload the prediction file, but this time to `test-dev` phase.

1.7 FAQs

[Coming Soon]

1.8 Hateful Memes

The Hateful Memes challenge is available at this link.

In MMF, we provide the starter code and baseline pretrained models for this challenge and the configurations used for training the reported baselines. For more details check this link.

In this tutorial, we provide steps for running training and evaluation with MMBT model on hateful memes dataset and generating submission file for the challenge. The same steps can be used for your own models.

1.8.1 Installation and Preparing the dataset

Follow the prerequisites for installation and dataset here.

1.8.2 Training and Evaluation

Training

For running training on train set, run the following command:

```
mmf_run config=projects/hateful_memes/configs/mmbt/defaults.yaml model=mmbt␣
↪dataset=hateful_memes training.run_type=train_val
```

This will train the `mmbt` model on the dataset and generate the checkpoints and best trained model (`mmbt_final.pth`) will be stored in the `./save` directory by default.

Evaluation

Next run evaluation on the validation set:

```
mmf_run config=projects/hateful_memes/configs/mmbt/defaults.yaml model=mmbt␣
↪dataset=hateful_memes training.run_type=val resume_file=./save/mmbt_final.pth
```

This will give you the performance of your model on the validation set. The metrics are AUROC, ACC, Binary F1 etc.

1.8.3 Predictions for Challenge

After we trained the model and evaluated on the validation set, we will generate the predictions on the test set. The prediction file should contain the following three columns:

- Meme identification number, `id`
- Probability that the meme is hateful, `proba`
- Binary label that the meme is hateful (1) or non-hateful (0), `label`

With MMF you can directly generate the predictions in the required submission format with the following command:

```
mmf_predict config=projects/hateful_memes/configs/mmbt/defaults.yaml model=mmbt␣
↪dataset=hateful_memes run_type=test
```

This command will output where the generated predictions csv file is stored.

1.8.4 Submission for Challenge

Next you can upload the generated csv file on DrivenData in their submissions page for Hateful Memes.

More details will be added once the challenge submission phase is live.

1.8.5 Building on top of MMF and Open Sourcing your code

To understand how you build on top of MMF for your own custom models and then open source your code, take a look at this example.

1.9 Terminology and Concepts

[Outdated] A new version of this will be uploaded soon

Authors: Amanpreet Singh

To develop on top of MMF, it is necessary to understand concepts and terminology used in MMF codebase. MMF has been very carefully designed from ground-up to be a multi-tasking framework. This means using MMF you can train on multiple datasets/datasets together.

To achieve this, MMF has few opinions about architecture of your research project. But, being generic means MMF abstracts a lot of concepts in its modules and it would be easy to develop on top of MMF once a developer understands these simple concepts. Major concepts and terminology in MMF that one needs to know in order to develop over MMF are as follows:

- *Tasks and Datasets*
- *Models*
- *Registry*
- *Configuration*
- *Processors*
- *Sample List*

1.9.1 Tasks and Datasets

In MMF, we have divided datasets into a set category of tasks. Thus, a task corresponds to a collection of datasets that belong to it. For example, VQA 2.0, VizWiz and TextVQA all belong VQA task. Each task and dataset has been assigned a unique key which is used to refer it in the command line arguments.

Following table shows the tasks and their datasets:

Task	Key	Datasets
VQA	vqa	VQA2.0, VizWiz, TextVQA, VisualGenome, CLEVR
Dialog	dialog	VisualDialog
Caption	captioning	MS COCO

Following table shows the inverse of the above table, datasets along with their tasks and keys:

Datasets	Key	Task	Notes
VQA 2.0	vqa2	vqa	
TextVQA	textvqa	vqa	
VizWiz	vizwiz	vqa	
VisualDialog	visdial	dialog	Coming soon!
VisualGenome	visual_genome	vqa	
CLEVR	clevr	vqa	
MS COCO	coco	captioning	

1.9.2 Models

Reference implementations for state-of-the-art models have been included to act as a base for reproduction of research papers and starting point of new research. MMF has been used in past for following papers:

- Towards VQA Models That Can Read (LoRRA model)
- VQA 2018 Challenge winner
- VizWiz 2018 Challenge winner

Similar to tasks and datasets, each model has been registered with a unique key for easy reference in configuration and command line arguments. Following table shows each model's key name and datasets it can be run on.

Model	Key	Datasets
LoRRA	lorra	vqa2, textvqa, vizwiz
Pythia	pythia	textvqa, vizwiz, vqa2, visual_genome
BAN	ban	textvqa, vizwiz, vqa2
BUTD	butd	coco
CNN LSTM	cnn_lstm	clevr

Note: BAN support is preliminary and hasn't been properly fine-tuned yet.

1.9.3 Registry

Registry acts as a central source of truth for MMF. Inspired from Redux's global store, useful information needed by MMF ecosystem is registered in the `registry`. Registry can be considered as a general purpose storage for information which is needed by multiple parts of the framework and acts source of information wherever that information is needed.

Registry also registers models, tasks, datasets etc. based on a unique key as mentioned above. Registry's functions can be used as decorators over the classes which need to be registered (for e.g. models etc.)

Registry object can be imported as the follow:

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

Find more details about Registry class in its documentation *common/registry*.

1.9.4 Configuration

As is necessary with research, most of the parameters/settings in MMF are configurable. MMF specific default values (`training`) are present in mmf/common/defaults/configs/base.yaml with detailed comments delineating the usage of each parameter.

For ease of usage and modularity, configuration for each dataset is kept separately in `mmf/common/defaults/configs/datasets/[task]/[dataset].yaml` where you can get `[task]` value for the dataset from the tables in *Tasks and Datasets* section.

The most dynamic part, model configuration are also kept separate and are the one which need to be defined by the user if they are creating their own models. We include configurations for the models included in the model zoo of MMF. For each model, there is a separate configuration for each dataset it can work on. See an example in configs/vqa/vqa2/pythia.yaml. The configuration in the configs folder are divided using the scheme `configs/[task]/[dataset]/[model].yaml`.

It is possible to include other configs into your config using `includes` directive. Thus, in MMF config above you can include `vqa2`'s config like this:

```
includes:
- common/defaults/configs/datasets/vqa/vqa2.yaml
```

Now, due to separate config per dataset this concept can be extended to do multi-tasking and include multiple dataset configs here.

`base.yaml` file mentioned above is always included and provides sane defaults for most of the training parameters. You can then specify the config of the model that you want to train using `--config [config_path]` option. The final config can be retrieved using `registry.get('config')` anywhere in your codebase. You can access the attributes from these configs by using `dot` notation. For e.g. if you want to get the value of maximum iterations, you can get that by `registry.get('config').training.max_updates`.

The values in the configuration can be overriden using two formats:

- Individual Override: For e.g. you want to use `DataParallel` to train on multiple GPUs, you can override the default value of `False` by passing arguments `training.data_parallel True` at the end your command. This will override that option on the fly.
- DemJSON based override: The above option gets clunky when you are trying to run the hyperparameters sweeps over model parameters. To avoid this, you can update a whole block using a demjson string. For e.g. to use early stopping as well update the patience, you can pass `--config_override "{training: {should_early_stop: True, patience: 5000}}"`. This demjson string is easier to generate programmatically than the individual override.

Note: It is always helpful to verify your config overrides and final configuration values that are printed to make sure you override the correct keys.

1.9.5 Processors

The main aim of processors is to keep data processing pipelines as similar as possible for different datasets and allow code reusability. Processors take in a dict with keys corresponding to data they need and return back a dict with processed data. This helps keep processors independent of the rest of the logic by fixing the signatures they require. Processors are used in all of the datasets to hand off the data processing needs. Learn more about processors in the *documentation for processors*.

1.9.6 Sample List

SampleList has been inspired from BBoxList in maskrcnn-benchmark, but is more generic. All datasets integrated with MMF need to return a Sample which will be collated into `SampleList`. Now, `SampleList` comes with a lot of handy functions which enable easy batching and access of things. For e.g. `Sample` is a dict with some keys. In `SampleList`, values for these keys will be smartly clubbed based on whether it is a tensor or a list and assigned back to that dict. So, end user gets these keys clubbed nicely together and can use them in their model. Models integrated with Pythia receive a `SampleList` as an argument which again makes the trainer unopinionated about the models as well as the datasets. Learn more about `Sample` and `SampleList` in their *documentation*.

1.10 Tutorial: Adding a dataset

[Outdated] A new version of this will be uploaded soon

# MMF

This is a tutorial on how to add a new dataset to MMF.

MMF is agnostic to kind of datasets that can be added to it. On high level, adding a dataset requires 4 main components.

- Dataset Builder
- Default Configuration
- Dataset Class
- Dataset's Metrics

In most of the cases, you should be able to inherit one of the existing datasets for easy integration. Let's start from the dataset builder

1.10.1 Dataset Builder

Builder creates and returns an instance of `mmf.datasets.base_dataset.BaseDataset` which is inherited from `torch.utils.data.dataset.Dataset`. Any builder class in MMF needs to be inherited from `mmf.datasets.base_dataset_builder.BaseDatasetBuilder`. `BaseDatasetBuilder` requires user to implement following methods after inheriting the class.

- `__init__(self):`

Inside this function call super().__init__("name") where "name" should your dataset's name like "vqa2".

- `load(self, config, dataset_type, *args, **kwargs)`

This function loads the dataset, builds an object of class inheriting `BaseDataset` which contains your dataset logic and returns it.

- `build(self, config, dataset_type, *args, **kwargs)`

This function actually builds the data required for initializing the dataset for the first time. For e.g. if you need to download some data for your dataset, this all should be done inside this function.

Finally, you need to register your dataset builder with a key to registry using `mmf.common.registry.registry.register_builder("key")`.

That's it, that's all you require for inheriting `BaseDatasetBuilder`.

Let's write down this using example of *CLEVR* dataset.

```
import json
import math
import os
import zipfile

from collections import Counter

from mmf.common.registry import registry
from mmf.datasets.base_dataset_builder import BaseDatasetBuilder
# Let's assume for now that we have a dataset class called CLEVRDataset
from mmf.datasets.builders.clevr.dataset import CLEVRDataset
from mmf.utils.general import download_file, get_mmf_root

@registry.register_builder("clevr")
class CLEVRBuilder(BaseDatasetBuilder):
    DOWNLOAD_URL = "https://s3-us-west-1.amazonaws.com/clevr/CLEVR_v1.0.zip"

    def __init__(self):
        # Init should call super().__init__ with the key for the dataset
        super().__init__("clevr")
```

(continues on next page)

(continued from previous page)

```
        self.writer = registry.get("writer")

        # Assign the dataset class
        self.dataset_class = CLEVRDataset

    def build(self, config, dataset):
        download_folder = os.path.join(
            get_mmf_root(), config.data_dir, config.data_folder
        )

        file_name = self.DOWNLOAD_URL.split("/")[-1]
        local_filename = os.path.join(download_folder, file_name)

        extraction_folder = os.path.join(download_folder, ".".join(file_name.split(".
↪")[:-1]))
        self.data_folder = extraction_folder

        # Either if the zip file is already present or if there are some
        # files inside the folder we don't continue download process
        if os.path.exists(local_filename):
            return

        if os.path.exists(extraction_folder) and \
            len(os.listdir(extraction_folder)) != 0:
            return

        self.writer.write("Downloading the CLEVR dataset now")
        download_file(self.DOWNLOAD_URL, output_dir=download_folder)

        self.writer.write("Downloaded. Extracting now. This can take time.")
        with zipfile.ZipFile(local_filename, "r") as zip_ref:
            zip_ref.extractall(download_folder)


    def load(self, config, dataset, *args, **kwargs):
        # Load the dataset using the CLEVRDataset class
        self.dataset = CLEVRDataset(
            config, dataset, data_folder=self.data_folder
        )
        return self.dataset

    def update_registry_for_model(self, config):
        # Register both vocab (question and answer) sizes to registry for easy access␣
↪to the
        # models. update_registry_for_model function if present is automatically␣
↪called by
        # MMF
        registry.register(
            self.dataset_name + "_text_vocab_size",
            self.dataset.text_processor.get_vocab_size(),
        )
        registry.register(
            self.dataset_name + "_num_final_outputs",
            self.dataset.answer_processor.get_vocab_size(),
        )
```

1.10.2 Default Configuration

Some things to note about MMF's configuration:

- Each dataset in MMF has its own default configuration which is usually under this structure `mmf/common/defaults/configs/datasets/[task]/[dataset].yaml` where `task` is the task your dataset belongs to.
- These dataset configurations can be then included by the user in their end config using `includes` directive
- This allows easy multi-tasking and management of configurations and user can also override the default configurations easily in their own config

So, for CLEVR dataset also, we will need to create a default configuration.

The config node is directly passed to your builder which you can then pass to your dataset for any configuration that you need for building your dataset.

Basic structure for a dataset configuration looks like below:

```
dataset_config:
    [dataset]:
        ... your config here
```

Here, is a default configuration for CLEVR needed based on our dataset and builder class above:

```
dataset_config:
    # You can specify any attributes you want, and you will get them as attributes
    # inside the config passed to the dataset. Check the Dataset implementation below.
    clevr:
        # Where your data is stored
        data_dir: ${env.data_dir}
        data_folder: CLEVR_v1.0
        # Any attribute that you require to build your dataset but are configurable
        # For CLEVR, we have attributes that can be passed to vocab building class
        build_attributes:
            min_count: 1
            split_regex: " "
            keep:
                - ";"
                - ","
            remove:
                - "?"
                - "."
        processors:
        # The processors will be assigned to the datasets automatically by MMF
        # For example if key is text_processor, you can access that processor inside
        # dataset object using self.text_processor
            text_processor:
                type: vocab
                params:
                    max_length: 10
                    vocab:
                        type: random
                        vocab_file: vocabs/clevr_question_vocab.txt
                # You can also specify a processor here
                preprocessor:
                    type: simple_sentence
                    params: {}
            answer_processor:
```

(continues on next page)

(continued from previous page)

```
                # Add your processor for answer processor here
                type: multi_hot_answer_from_vocab
                params:
                    num_answers: 1
                    # Vocab file is relative to [data_dir]/[data_folder]
                    vocab_file: vocabs/clevr_answer_vocab.txt
                    preprocessor:
                        type: simple_word
                        params: {}
```

For processors, check *mmf.datasets.processors* to understand how to create a processor and different processors that are already available in MMF.

1.10.3 Dataset Class

Next step is to actually build a dataset class which inherits *BaseDataset* so it can interact with PyTorch dataloaders. Follow the steps below to inherit and create your dataset's class.

- Inherit *mmf.datasets.base_dataset.BaseDataset*
- Implement `__init__(self, config, dataset)`. Call parent's init using `super().__init__("name", config, dataset)` where "name" is the string representing the name of your dataset.
- Implement `__getitem__(self, idx)`, our replacement for normal `__getitem__(self, idx)` you would implement for a torch dataset. This needs to return an object of class :class:Sample.
- Implement `__len__(self)` method, which represents size of your dataset.
- [Optional] Implement `load_item(self, idx)` if you need to load something or do something else with data and then call it inside `__getitem__`.

```python
import os
import json

import numpy as np
import torch

from PIL import Image

from mmf.common.registry import registry
from mmf.common.sample import Sample
from mmf.datasets.base_dataset import BaseDataset
from mmf.utils.general import get_mmf_root
from mmf.utils.text import VocabFromText, tokenize


class CLEVRDataset(BaseDataset):
    def __init__(self, config, dataset, data_folder=None, *args, **kwargs):
        super().__init__("clevr", config, dataset)
        self._data_folder = data_folder
        self._data_dir = os.path.join(get_mmf_root(), config.data_dir)

        if not self._data_folder:
            self._data_folder = os.path.join(self._data_dir, config.data_folder)

        if not os.path.exists(self._data_folder):
```

(continues on next page)

(continued from previous page)

```
            raise RuntimeError(
                "Data folder {} for CLEVR is not present".format(self._data_folder)
            )

        # Check if the folder was actually extracted in the subfolder
        if config.data_folder in os.listdir(self._data_folder):
            self._data_folder = os.path.join(self._data_folder, config.data_folder)

        if len(os.listdir(self._data_folder)) == 0:
            raise RuntimeError("CLEVR dataset folder is empty")

        self._load()

    def _load(self):
        self.image_path = os.path.join(self._data_folder, "images", self._dataset_
↪type)

        with open(
            os.path.join(
                self._data_folder,
                "questions",
                "CLEVR_{}_questions.json".format(self._dataset_type),
            )
        ) as f:
            self.questions = json.load(f)["questions"]
            self._build_vocab(self.questions, "question")
            self._build_vocab(self.questions, "answer")

    def __len__(self):
        # __len__ tells how many samples are there
        return len(self.questions)

    def _get_vocab_path(self, attribute):
        return os.path.join(
            self._data_dir, "vocabs",
            "{}_{}_vocab.txt".format(self.dataset_name, attribute)
        )

    def _build_vocab(self, questions, attribute):
        # This function builds vocab for questions and answers but not required for␣
↪the
        # tutorial
        ...

    def __getitem__(self, idx):
        # Get item is like your normal __getitem__ in PyTorch Dataset. Based on id
        # return a sample. Check VQA2Dataset implementation if you want to see how
        # to do caching in MMF
        data = self.questions[idx]

        # Each call to __getitem__ from dataloader returns a Sample class object which
        # collated by our special batch collator to a SampleList which is basically
        # a attribute based batch in layman terms
        current_sample = Sample()

        question = data["question"]
        tokens = tokenize(question, keep=[";", ","], remove=["?", "."])
```

(continues on next page)

(continued from previous page)

```
        # This processors are directly assigned as attributes to dataset based on the␣
↪config
        # we created above
        processed = self.text_processor({"tokens": tokens})
        # Add the question as text attribute to the sample
        current_sample.text = processed["text"]

        processed = self.answer_processor({"answers": [data["answer"]]})
        # Now add answers and then the targets. We normally use "targets" for what
        # should be the final output from the model in MMF
        current_sample.answers = processed["answers"]
        current_sample.targets = processed["answers_scores"]

        image_path = os.path.join(self.image_path, data["image_filename"])
        image = np.true_divide(Image.open(image_path).convert("RGB"), 255)
        image = image.astype(np.float32)
        # Process and add image as a tensor
        current_sample.image = torch.from_numpy(image.transpose(2, 0, 1))

        # Return your sample and MMF will automatically convert it to SampleList␣
↪before
        # passing to the model
        return current_sample
```

1.10.4 Metrics

For your dataset to be compatible out of the box, it is a good practice to also add the metrics your dataset requires. All metrics for now go inside `MMF/modules/metrics.py`. All metrics inherit *BaseMetric* and implement a function `calculate` with signature `calculate(self, sample_list, model_output, *args, **kwargs)` where `sample_list` (*SampleList*) is the current batch and `model_output` is a dict return by your model for current `sample_list`. Normally, you should define the keys you want inside `model_output` and `sample_list`. Finally, you should register your metric to registry using `@registry.register_metric('[key]')` where '[key]' is the key for your metric. Here is a sample implementation of accuracy metric used in CLEVR dataset:

These are the common steps you need to follow when you are adding a dataset to MMF.

1.11 Tutorial: Late Fusion

[Coming Soon]

1.12 Tutorial: Detect Hate Speech with MMFBERT

[Coming Soon]

1.13 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 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 decoder: `@registry.register_decoder`

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_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_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_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")
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")
```

1.14 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

def __getitem__(self, key): return self.__dict__[key] field present in the `SampleList` currently.

Returns:

def __getitem__(self, key):

return self.__dict__[key] int: Size of the batch in `SampleList`.

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]

1.15 models.base_model

Models built on top of Pythia need to inherit `BaseModel` class and adhere to some 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:

```python
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*)

For integration with Pythia'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.

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.

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 key when pretrained model is being load. 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

1.16 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.

```python
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.**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.`**`CrossEntropyLoss`**(*params=None*)

`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.`**`Losses`**(*loss_list*)
`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 instanttions of each loss passed in config

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

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.**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.**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.

1.17 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:

```python
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**

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

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.`**`BinaryF1`**`(*args, **kwargs)`

Metric for calculating Binary F1.

Key: `binary_f1`

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.`**`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.`**`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.`**`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.`**`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.**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.**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.**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.**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

1.18 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):
        ...
```

(continues on next page)

(continued from previous page)

```
        return Dataset()

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

class mmf.datasets.base_dataset_builder.**BaseDatasetBuilder**(*dataset_name*)

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.

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`.

1.19 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

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*)

1.20 datasets.processors

class `mmf.datasets.processors.`**`BaseProcessor`**(*config*, **args*, ***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.`**`Processor`**(*config*, **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 procesor.

class `mmf.datasets.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:

```
task_attributes:
    vqa:
        vqa2:
            processors:
              text_processor:
                type: vocab
                params:
                  max_length: 14
                  vocab:
                    type: intersected
                    embedding_name: glove.6B.300d
                    vocab_file: 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.**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.**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.**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.`**`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.`**`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.**SimpleWordProcessor**(**args, **kwargs*)
Tokenizes a word and processes it.

tokenizer
Type of tokenizer to be used.

Type function

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

tokenizer
Type of tokenizer to be used.

Type function

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

_truncate_seq_pair(*tokens_a*, *tokens_b*, *max_length*)
Truncates a sequence pair in place to the maximum length.

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

1.21 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]

1.22 Sample Model Config

```yaml
model_config:
  mmbt:
    # Either pretraining or classification
    training_head_type: pretraining
    bert_model_name: bert-base-uncased
    direct_features_input: false
    freeze_text: false
    freeze_modal: false
    freeze_complete_base: false
    finetune_lr_multiplier: 1
    # Dimension of the embedding finally returned by the modal encoder
    modal_hidden_size: 2048
    # Dimension of the embedding finally returned by the text encoder
    text_hidden_size: 768
    # Used when classification head is activated
    num_labels: 2
    modal_encoder:
      type: resnet152
      params:
        pretrained: true
        pool_type: avg
        num_output_features: 1

    use_modal_start_token: true
    use_modal_end_token: true
    text_encoder:
      type: transformer
      params:
        bert_model_name: ${model_config.mmbt.bert_model_name}
        # Options below can be overridden to update the bert configuration used
        # to initialize the bert encoder. If some option is missing or
        # if you are using an encoder different then BERT, add extra parameters
        # to your projects configuration file under model_config.mmbt.
        # Those options will automatically override the options for your transformer
        # encoder's configuration. For e.g. vocab_size is missing here, just add
        # vocab_size: x to update the size of the vocabulary with which encoder is
        # initialized. If you update the default values, the transformer you
        # will get will be initialized from scratch.
```

(continues on next page)

(continued from previous page)

```yaml
        hidden_size: 768
        num_hidden_layers: 12
        num_attention_heads: 12
        output_attentions: false
        output_hidden_states: false
```

1.23 Sample Dataset Config

```yaml
dataset_config:
  hateful_memes:
    data_dir: ${env.data_dir}/datasets
    depth_first: false
    fast_read: false
    use_images: true
    use_features: false
    images:
      train:
      - hateful_memes/defaults/images/
      val:
      - hateful_memes/defaults/images/
      test:
      - hateful_memes/defaults/images/
    features:
      train:
      - hateful_memes/defaults/features/detectron.lmdb
      val:
      - hateful_memes/defaults/features/detectron.lmdb
      test:
      - hateful_memes/defaults/features/detectron.lmdb
    annotations:
      train:
      - hateful_memes/defaults/annotations/train.jsonl
      val:
      - hateful_memes/defaults/annotations/dev.jsonl
      test:
      - hateful_memes/defaults/annotations/test.jsonl
    max_features: 100
    processors:
      text_processor:
        type: vocab
        params:
          max_length: 14
          vocab:
            type: intersected
            embedding_name: glove.6B.300d
            vocab_file: hateful_memes/defaults/extras/vocabs/vocabulary_100k.txt
          preprocessor:
            type: simple_sentence
            params: {}
      bbox_processor:
        type: bbox
        params:
          max_length: 50
      image_processor:
```

(continues on next page)

(continued from previous page)

```yaml
        type: torchvision_transforms
        params:
          transforms:
            - type: Resize
              params:
                size: [256, 256]
            - type: CenterCrop
              params:
                size: [224, 224]
            - ToTensor
            - GrayScaleTo3Channels
            - type: Normalize
              params:
                mean: [0.46777044, 0.44531429, 0.40661017]
                std: [0.12221994, 0.12145835, 0.14380469]
    return_features_info: false


evaluation:
  predict_file_format: csv
```

1.24 Sample User Config

```yaml
includes:
- configs/models/mmbt/classification.yaml
- configs/datasets/hateful_memes/bert.yaml

scheduler:
  type: warmup_linear
  params:
    num_warmup_steps: 2000
    num_training_steps: ${training.max_updates}

optimizer:
  type: adam_w
  params:
    lr: 1e-5
    eps: 1e-8

evaluation:
    metrics:
    - accuracy
    - binary_f1
    - roc_auc

training:
  batch_size: 32
  lr_scheduler: true
  max_updates: 22000
  early_stop:
    criteria: hateful_memes/roc_auc
    minimize: false

checkpoint:
```

(continues on next page)

(continued from previous page)

```
  pretrained_state_mapping:
    bert: bert
```

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index