MMF Documentation

Release 0.3

Facebook AI Research

May 14, 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 1.4.

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 Note that, pip packages are always outdated compared to installing from source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

[Outdated] A new version of this will be uploaded soon

Authors: Amanpreet Singh

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

1.2.1 Demo

1. VQA
2. Captioning

1.2.2 Citation

If you use MMF in your work, please cite:

```
@inproceedings{Singh2019TowardsVM,
  title={Towards VQA Models That Can Read},
  author={Singh, Amanpreet and Natarajan, Vivek and Shah, Meet and Jiang, Yu and Chen,
↪ Xinlei and Batra, Dhruv and Parikh, Devi and Rohrbach, Marcus},
  booktitle={Proceedings of the IEEE Conference on Computer Vision and Pattern␣
↪Recognition},
  year={2019}
}
```

and

```
@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}
}
```

1.2.3 Installation

1. Clone MMF repository

```
git clone https://github.com/facebookresearch/mmf ~/mmf
```

1. Install dependencies and setup

```
cd ~/mmf
python setup.py develop
```

Note:

1. If you face any issues with the setup, check the Troubleshooting/FAQ section below.
2. You can also create/activate your own conda environments before running above commands.

1.2.4 Getting Data

Datasets currently supported in MMF require two parts of data, features and ImDB. Features correspond to pre-extracted object features from an object detector. ImDB is the image database for the datasets which contains information such as questions and answers in case of TextVQA.

For TextVQA, we need to download features for OpenImages' images which are included in it and TextVQA 0.5 ImDB. We assume that all of the data is kept inside `data` folder under `mmf` root folder. Table in bottom shows corresponding features and ImDB links for datasets supported in MMF.

```
cd ~/mmf;
# Create data folder
mkdir -p data && cd data;

# Download and extract the features
wget https://dl.fbaipublicfiles.com/pythia/features/open_images.tar.gz
tar xf open_images.tar.gz

# Get vocabularies
wget http://dl.fbaipublicfiles.com/pythia/data/vocab.tar.gz
tar xf vocab.tar.gz

# Download detectron weights required by some models
wget http://dl.fbaipublicfiles.com/pythia/data/detectron_weights.tar.gz
tar xf detectron_weights.tar.gz

# Download and extract ImDB
mkdir -p imdb && cd imdb
wget https://dl.fbaipublicfiles.com/pythia/data/imdb/textvqa_0.5.tar.gz
tar xf textvqa_0.5.tar.gz
```

1.2.5 Training

Once we have the data in-place, we can start training by running the following command:

```
cd ~/mmf;
python tools/run.py --datasets textvqa --model lorra --config \
configs/vqa/textvqa/lorra.yaml
```

1.2.6 Inference

For running inference or generating predictions for EvalAI, we can download a corresponding pretrained model and then run the following commands:

```
cd ~/mmf/data
mkdir -p models && cd models;
wget https://dl.fbaipublicfiles.com/pythia/pretrained_models/textvqa/lorra_best.pth
cd ../..
python tools/run.py --datasets textvqa --model lorra --config \
configs/vqa/textvqa/lorra.yaml --resume_file data/models/lorra_best.pth \
--evalai_inference 1 --run_type inference
```

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 Pythia.

1.2.7 Troubleshooting/FAQs

1. If `setup.py` causes any issues, please install fastText first directly from the source and then run `python setup.py develop`. To install fastText run following commands:

```
git clone https://github.com/facebookresearch/fastText.git
cd fastText
pip install -e .
```

1.2.8 Tasks and Datasets

Dataset	Key	Task	ImDB Link	Features Link	Features checksum	Notes
TextVQA	textvqa	vqa	TextVQA 0.5 ImDB	OpenImages	*b22e80997b2580edaf08d7e3a896e324*	
VQA 2.0	vqa2	vqa	VQA 2.0 ImDB	COCO	*ab7947b04f3063c774b87dfbf4d0e981*	
VizWiz	vizwiz	vqa	VizWiz ImDB	VizWiz	*9a28d6a9892dda8519d03fba52fb899f*	
Visual-Dialog	visdial	dialog	Coming soon!	Coming soon!	Coming soon!	
VisualGenome	visual_genome	vqa	Automatically downloaded	Automatically downloaded	Coming soon!	Also supports scene graphs
CLEVR	clevr	vqa	Automatically downloaded	Automatically downloaded		
MS COCO	coco	captioning	COCO Caption	COCO	*ab7947b04f3063c774b87dfbf4d0e981*	

After downloading the features, verify the download by checking the md5sum using

```
echo "<checksum>  <dataset_name>.tar.gz" | md5sum -c -
```

1.2.9 Next steps

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

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

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 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.

1.5.1 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
> 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.

1.5.2 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.

1.6 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.

1.6.1 Installation and Preparing the dataset

Follow the prerequisites for installation and dataset here.

1.6.2 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.

1.6.3 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.

1.6.4 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.

1.6.5 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.

1.7 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*

1.7.1 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	

1.7.2 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.

1.7.3 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*.

1.7.4 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.

1.7.5 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*.

1.7.6 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*.

1.8 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

1.8.1 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")
        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(
```

(continues on next page)

(continued from previous page)

```
            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(),
        )
```

1.8.2 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: " "
```

(continues on next page)

(continued from previous page)

```
            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:
                # 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.

1.8.3 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__`.

```
import os
import json
```

(continues on next page)

(continued from previous page)

```
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):
            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",
```

(continues on next page)

(continued from previous page)

```
            "{}_{}_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=["?", "."])

        # 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
```

1.8.4 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.

1.9 Tutorial: Late Fusion

[Coming Soon]

1.10 Tutorial: Detect Hate Speech with MMFBERT

[Coming Soon]

1.11 common.registry

1.12 common.sample

1.13 models.base_model

1.14 modules.losses

1.15 modules.metrics

1.16 datasets.base_dataset_builder

1.17 datasets.base_dataset

1.18 datasets.processors

1.19 utils.text

CHAPTER 2

Indices and tables

- genindex
- modindex
- search