Pythia Documentation

Release 0.3

Facebook AI Research

Aug 26, 2019

Tutorials

Pythia is a modular framework for supercharging vision and language research built on top of PyTorch.

CHAPTER 1

Quickstart

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 Pythia.

1.1 Demo

1. Pythia VQA
2. BUTD Captioning

1.2 Installation

1. Clone Pythia repository

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

1. Install dependencies and setup

```
cd ~/pythia
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.

1.3 Getting Data

Datasets currently supported in Pythia 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 `pythia` root folder. Table in bottom shows corresponding features and ImDB links for datasets supported in pythia.

```
cd ~/pythia;
# 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
```

1.4 Training

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

```
cd ~/pythia;
python tools/run.py --tasks vqa --datasets textvqa --model lorra --config \
configs/vqa/textvqa/lorra.yml
```

1.5 Inference

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

```
cd ~/pythia/data
mkdir -p models && cd models;
wget https://dl.fbaipublicfiles.com/pythia/pretrained_models/textvqa/lorra_best.pth
cd ../..
python tools/run.py --tasks vqa --datasets textvqa --model lorra --config \
configs/vqa/textvqa/lorra.yml --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.

1.6 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 .
```

1.7 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 -
```

1.8 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

1.9 Citation

If you use Pythia 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}
}
```

CHAPTER 2

Terminology and Concepts

Authors: Amanpreet Singh

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

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

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

2.1 Tasks and Datasets

In Pythia, 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	

2.2 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. Pythia 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	textvqa, vizwiz
Pythia	pythia	textvqa, vizwiz, vqa2, visual_genome
BAN	ban	textvqa, vizwiz, vqa2
BUTD	butd	coco

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

2.3 Registry

Registry acts as a central source of truth for Pythia. Inspired from Redux's global store, useful information needed by Pythia 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 pythia.common.registry import registry
```

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

2.4 Configuration

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

For ease of usage and modularity, configuration for each dataset is kept separately in `pythia/common/defaults/configs/tasks/[task]/[dataset].yml` 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 Pythia. For each model, there is a separate configuration for each dataset it can work on. See an example in configs/vqa/vqa2/pythia.yml. The configuration in the configs folder are divided using the scheme `configs/[task]/[dataset]/[model].yml`.

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

```
includes:
- common/defaults/configs/tasks/vqa/vqa2.yml
```

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

`base.yml` 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_parameters.max_iterations`.

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_parameters.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_parameters: {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.

2.5 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*.

2.6 Sample List

SampleList has been inspired from BBoxList in maskrcnn-benchmark, but is more generic. All datasets integrated with Pythia 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*.

CHAPTER 3

Features

Pythia 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 Pythia to **bootstrap** for your next vision and language multimodal research project.

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

CHAPTER 4

Adding a dataset

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

Pythia 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
- [Optional] Task specification

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

4.1 Dataset Builder

Builder creates and returns an instance of *pythia.tasks.base_dataset.BaseDataset* which is inherited from `torch.utils.data.dataset.Dataset`. Any builder class in Pythia needs to be inherited from *pythia.tasks.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, dataset_type, config, *args, **kwargs)`

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

- `_build(self, dataset_type, config, *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 `pythia.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 pythia.common.registry import registry
from pythia.tasks.base_dataset_builder import BaseDatasetBuilder
# Let's assume for now that we have a dataset class called CLEVRDataset
from pythia.tasks.vqa.clevr.dataset import CLEVRDataset
from pythia.utils.general import download_file, get_pythia_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, dataset_type, config):
        download_folder = os.path.join(
            get_pythia_root(), config.data_root_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)
```

(continues on next page)

(continued from previous page)

```
        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, dataset_type, config, *args, **kwargs):
        # Load the dataset using the CLEVRDataset class
        self.dataset = CLEVRDataset(
            dataset_type, config, 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
        # pythia
        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(),
        )
```

4.2 Default Configuration

Some things to note about Pythia's configuration:

- Each dataset in Pythia has its own default configuration which is usually under this structure `pythia/commmon/defaults/configs/tasks/[task]/[dataset].yml` 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:

```
task_attributes:
    [task]:
        datasets:
        - [dataset]
        dataset_attributes:
            [dataset]:
                ... your config here
```

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

```yaml
task_attributes:
    vqa:
        datasets:
        - clevr
        dataset_attributes:
            # 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_root_dir: ../data
                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␣
↪Pythia
                # 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_root_dir]/[data_folder]
                            vocab_file: vocabs/clevr_answer_vocab.txt
                            preprocessor:
                                type: simple_word
                                params: {}
training_parameters:
    monitored_metric: clevr_accuracy
    metric_minimize: false
```

Extra field that we have added here is `training_parameters` which specify the dataset specific training pa-

rameters and will be merged with the rest of the training parameters coming from user's config. Your metrics are normally stored in registry as `[dataset]_[metric_key]`, so to monitor accuracy on CLEVR, you need to set it as `clevr_accuracy` and we need to maximize it, we set `metric_minimize` to `false`.

For processors, check *pythia.tasks.processors* to understand how to create a processor and different processors that are already available in Pythia.

4.3 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 *pythia.tasks.base_dataset.BaseDataset*
- Implement `__init__(self, dataset_type, config)`. Call parent's init using `super().__init__("name", dataset_type, config)` where "name" is the string representing the name of your dataset.
- Implement `get_item(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 `get_item`.

```
import os
import json

import numpy as np
import torch

from PIL import Image

from pythia.common.registry import registry
from pythia.common.sample import Sample
from pythia.tasks.base_dataset import BaseDataset
from pythia.utils.general import get_pythia_root
from pythia.utils.text_utils import VocabFromText, tokenize


class CLEVRDataset(BaseDataset):
    def __init__(self, dataset_type, config, data_folder=None, *args, **kwargs):
        super().__init__("clevr", dataset_type, config)
        self._data_folder = data_folder
        self._data_root_dir = os.path.join(get_pythia_root(), config.data_root_dir)

        if not self._data_folder:
            self._data_folder = os.path.join(self._data_root_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):
```

(continues on next page)

(continued from previous page)

```
            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_root_dir, "vocabs",
            "{}_{}_vocab.txt".format(self._name, attribute)
        )

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

    def get_item(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 Pythia
        data = self.questions[idx]

        # Each call to get_item 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
```

(continues on next page)

(continued from previous page)

```
        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 Pythia
        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 Pythia will automatically convert it to SampleList␣
↪before
        # passing to the model
        return current_sample
```

4.4 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 `pythia/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:

4.5 [Optional] Task Specification

This optional step is required in case you are adding a new task type to Pythia. Check implementation of VQATask to understand an implementation of task specification. In most cases, you don't need to do this.

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

CHAPTER 5

Pretrained Models

Performing inference using pretrained models in Pythia 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].yml`. For example, config file for `vqa2 train_and_val` should be `configs/vqa/vqa2/pythia_train_and_val.yml`. Now to run inference for EvalAI, run the following command.

```
cd ~/pythia/data
mkdir -p models && cd models;
# Download the pretrained model
wget [link]
cd ../..;
python tools/run.py --tasks [task] --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.yml`. 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.yml` config that supports beam search decoding.

CHAPTER 6

Challenge Participation

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

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

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 ~/pythia/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 tasks, datasets and model with corresponding key for other pretrained models
python tools/run.py --tasks vqa --datasets textvqa --model lorra --config configs/vqa/
↪textvqa/lorra.yml \
--run_type val --evalai_inference 1 --resume_file data/models/lorra_best.pth
```

In the printed log, Pythia 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 Pythia to beat LoRRA in challenge.

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 ~/pythia/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 tasks, datasets and model with corresponding key for other pretrained models
python tools/run.py --tasks vqa --datasets vqa2 --model pythia --config configs/vqa/
↪vqa2/pythia.yml \
--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.

CHAPTER 7

common.registry

Registry is central source of truth in Pythia. 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 pythia.common.registry import registry
```

Various decorators for registry different kind of classes with unique keys

- Register a task: `@registry.register_task`
- 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 model: `@registry.register_model`
- Register a processor: `@registry.register_processor`
- Register a optimizer: `@registry.register_optimizer`
- Register a scheduler: `@registry.register_scheduler`

class pythia.common.registry.**Registry**

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

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 retreived.
- **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 Pythia's internal operations. Default: False

Usage:

```
from pythia.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 pythia.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 pythia.common.registry import registry
from pythia.tasks.base_dataset_builder import BaseDatasetBuilder


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

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

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

Usage:

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


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

(continues on next page)

(continued from previous page)

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

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

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

Usage:

```
from pythia.common.registry import registry
from pythia.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 pythia.common.registry import registry
from pythia.tasks.processors import BaseProcessor

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

classmethod register_task(*name*)
Register a task to registry with key 'name'

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

Usage:

```
from pythia.common.registry import registry
from pythia.tasks.base_task import BaseTask


@registry.register_task("vqa")
class VQATask(BaseTask):
    ...
```

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

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

Usage:

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


@registry.register_trainer("custom_trainer")
```

(continues on next page)

(continued from previous page)

```
class CustomTrainer():
    ...
```

classmethod unregister(*name*)

Remove an item from registry with key 'name'

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

Usage:

```
from pythia.common.registry import registry

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

CHAPTER 8

common.sample

`Sample` and `SampleList` are data structures for arbitary data returned from a dataset. To work with Pythia, 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 arbitary 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 pythia.common.sample.**Sample**(*init_dict={}*)

Sample represent some arbitary data. All datasets in Pythia 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 pythia.common.sample.**SampleList**(*samples=[]*)

`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 field present in the `SampleList` currently.

Returns Size of the batch in `SampleList`.

Return type int

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

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

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

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

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

Return type *SampleList*

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

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

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

Return type *SampleList*

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*

CHAPTER 9

models.base_model

Models built on top of Pythia need to inherit `BaseModel` class and adhere to some format. To create a model for Pythia, 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 and Metrics are automatically calculated by the `BaseModel` class and added to this dict if not present.

Example:

```
import torch

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


@registry.register("pythia")
class Pythia(BaseModel):
    # config is model_attributes 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 pythia.models.base_model.**BaseModel**(*config*)

For integration with Pythia's trainer, datasets and other feautures, 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** (*ConfigNode*) – `model_attributes` 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.

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_and_metrics()

Initializes loss and metrics for the model based `losses` key and `metrics` keys. Automatically called by Pythia internally after building the model.

CHAPTER 10

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 pythia using the following example.

```
from pythia.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_attributes:
    some_model:
        losses:
            - type: custom
            - params: {}
```

class pythia.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 pythia.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 pythia.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 pythia.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 pythia.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 pythia.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_attributes*. `loss_list` needs to be a list for each separate loss containing *type* and *params* attributes.

Parameters **`loss_list`** (*`List[ConfigNode]`*) – 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` pythia.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 pythia.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 pythia.modules.losses.**PythiaLoss**(*params={}*)

Internal Pythia 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/logit_bce": 27.4}`, in case *logit_bce* is used and SampleList is from *val* set.

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

Note: Since, `PythiaLoss` 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 pythia.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 pythia.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 pythia.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.

CHAPTER 11

modules.metrics

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

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

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

Example:

```
import torch

from pythia.common.registry import registry
from pythia.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_attributes:
    pythia:
        metrics:
        - type: some
          params:
            some_param: a
```

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

Base class to be inherited by all metrics registered to Pythia. 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 pythia.modules.metrics.**CaptionBleu4Metric**

Metric for calculating caption accuracy using BLEU4 Score.

Key: caption_bleu4

bleu_score = <module 'nltk.translate.bleu_score' from '/home/docs/checkouts/readthedoc

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 pythia.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 pythia.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 pythia.modules.metrics.**Metrics**(*metric_list*)
Internally used by Pythia, 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** (*List[ConfigNode]*) – List of ConfigNodes where each ConfigNode specifies name and parameters of the metrics used.

class pythia.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 pythia.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 pythia.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 pythia.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 pythia.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

CHAPTER 12

tasks.base_dataset_builder

In Pythia, 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 Pythia.

`_load` is used to load a dataset from specific path. `_load` needs to return an instance of subclass of `pythia.tasks.base_dataset.BaseDataset`.

See complete example for `VQA2DatasetBuilder` here.

Example:

```
from torch.utils.data import Dataset

from pythia.tasks.base_dataset_builder import BaseDatasetBuilder
from pythia.common.registry import registry

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

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

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

class pythia.tasks.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(*dataset_type*, *config*, **args*, ***kwargs*)
This is used to build a dataset first time. Implement this method in your child dataset builder class.

Parameters

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

_load(*dataset_type*, *config*, **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

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

Returns Dataset containing data to be trained on

Return type dataset (*BaseDataset*)

build(*dataset_type*, *config*, **args*, ***kwargs*)
Similar to load function, used by Pythia 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

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

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

load(*dataset_type*, *config*, **args*, ***kwargs*)
Main load function use by Pythia. This will internally call `_load` function. Calls `init_processors` and `try_fast_read` on the dataset returned from `_load`

Parameters

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

Returns Dataset containing data to be trained on

Return type dataset (*BaseDataset*)

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

CHAPTER 13

tasks.base_dataset

class pythia.tasks.base_dataset.**BaseDataset**(*name*, *dataset_type*, *config={}*)

Base class for implementing a dataset. Inherits from PyTorch's Dataset class but adds some custom functionality on top. Instead of __getitem__ you have to implement get_item here. Processors mentioned in the configuration are automatically initialized for the end user.

Parameters

- **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** (*ConfigNode*) – Configuration for the current dataset

get_item(*idx*)

Basically, __getitem__ of a torch dataset.

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

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 overriden 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*)

CHAPTER 14

tasks.base_task

Tasks come above datasets in hierarchy level. In case you want to implement a new task, you need to inherit `BaseTask` class. You need to implement `_get_available_datasets` and `_preprocess_item` functions to complete the implementation. You can check the source to see if you need to override any other methods like `prepare_batch`.

Check example of `VQATask` here.

Example:

```
from pythia.tasks.base_task import BaseTask
from pythia.common.registry import registry


@registry.register_task("my")
class MyTask(BaseTask):
    def __init__(self):
        super().__init__("my")

    def _get_available_datasets(self):
        return ["my"]

    def _preprocess_item(self):
        item.text = None
        return item
```

class pythia.tasks.base_task.**BaseTask**(*task_name*)

BaseTask that task classes need to inherit in order to create a new task.

Users must implement `_get_available_datasets` and `_preprocess_item` in order to complete implementation.

Parameters **task_name** (*str*) – Name of the task with which it will be registered

_get_available_datasets()

Set available datasets for this task here. Override in your child task class Temporary solution, later we will use decorators to easily register datasets with a task

Returns List - List of available datasets for this particular task

_init_args(*parser*)
Override this function to add extra parameters to parser in your child task class.

Parameters **parser** (*ArgumentParser*) – Original parser object passed from the higher level classes like trainer

Returns Description of returned object.

Return type type

_preprocess_item(*item*)
Preprocess an item to be returned from __getitem__. Override in your child task class, so you have control on what you are returning

Parameters **item** (Sample) – Sample returned by a particular dataset

Returns Preprocessed item

Return type *Sample*

clean_config(*config*)
Override this in case you want to clean the config you updated earlier in update_registry_for_model

update_registry_for_model(*config*)
Use this if there is some specific configuration required by model which must be inferred at runtime.

CHAPTER 15

tasks.processors

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

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

To create a new processor, follow these steps:

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

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

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

Config:

```
task_attributes:
    vqa:
        datasets:
        - vqa2
        dataset_attributes:
            vqa2:
                processors:
                  text_processor:
                    type: vocab
                    params:
                      max_length: 14
                      vocab:
                        type: intersected
```

(continues on next page)

(continued from previous page)

```
                    embedding_name: glove.6B.300d
                    vocab_file: vocabs/vocabulary_100k.txt
                  answer_processor:
                    type: vqa_answer
                    params:
                      num_answers: 10
                      vocab_file: vocabs/answers_vqa.txt
                      preprocessor:
                        type: simple_word
                        params: {}
```

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

Example:

```
from pythia.common.registry import registry
from pythia.tasks.processors import BaseProcessor


class MyProcessor(BaseProcessor):
    def __init__(self, config, *args, **kwargs):
        return

    def __call__(self, item, *args, **kwargs):
        text = item['text']
        text = [t.strip() for t in text.split(" ")]
        return {"text": text}
```

class pythia.tasks.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 pythia.tasks.processors.**BaseProcessor**(*config*, **args*, ***kwargs*)
Every processor in Pythia needs to inherit this class for compatability with Pythia. End user mainly needs to implement __call__ function.

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

class pythia.tasks.processors.**CaptionProcessor**(*config*, **args*, ***kwargs*)
Processes a caption with start, end and pad tokens and returns raw string.

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

class pythia.tasks.processors.**FastTextProcessor**(*config*, **args*, ***kwargs*)
FastText processor, similar to GloVe processor but returns FastText vectors.

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

class pythia.tasks.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** (*ConfigNode*) – Configuration parameters for GloVe same as *VocabProcessor()*.

class pythia.tasks.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 pythia.tasks.processors.**Processor**(*config*, **args*, ***kwargs*)
Wrapper class used by Pythia 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** (*ConfigNode*) – ConfigNode containing `type` of the processor to be initialized and `params` of that procesor.

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

tokenizer
Type of tokenizer to be used.

Type function

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

tokenizer
Type of tokenizer to be used.

Type function

class pythia.tasks.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** (*ConfigNode*) – 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 pythia.tasks.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** (*ConfigNode*) – 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 pythia.tasks.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** (*ConfigNode*) – 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

CHAPTER 16

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

Index

Symbols

A

B

C

F

G

I

L

M

N

P

R

S

T

U

V

W