OpenVQA

Sep 03, 2021

The Basics

OpenVQA is a general platform for visual question ansering (VQA) research, with implementing state-of-the-art approaches on different benchmark datasets. Supports for more methods and datasets will be updated continuously.

CHAPTER 1

Installation

This page provides basic prerequisites to run OpenVQA, including the setups of hardware, software, and datasets.

1.1 Hardware & Software Setup

A machine with at least **1 GPU (>= 8GB)**, **20GB memory** and **50GB free disk space** is required. We strongly recommend to use a SSD drive to guarantee high-speed I/O.

The following packages are required to build the project correctly.

- Python >= 3.5
- Cuda >= 9.0 and cuDNN
- PyTorch >= 0.4.1 with CUDA (**PyTorch 1.x is also supported**).
- SpaCy and initialize the GloVe as follows:

```
$ pip install -r requirements.txt
$ wget https://github.com/explosion/spacy-models/releases/download/en_vectors_web_lg-
↪2.1.0/en_vectors_web_lg-2.1.0.tar.gz -O en_vectors_web_lg-2.1.0.tar.gz
$ pip install en_vectors_web_lg-2.1.0.tar.gz
```

1.2 Dataset Setup

The following datasets should be prepared before running the experiments.

Note that if you only want to run experiments on one specific dataset, you can focus on the setup for that and skip the rest.

1.2.1 VQA-v2

- Image Features

The image features are extracted using the bottom-up-attention strategy, with each image being represented as an dynamic number (from 10 to 100) of 2048-D features. We store the features for each image in a `.npz` file. You can prepare the visual features by yourself or download the extracted features from OneDrive or BaiduYun. The downloaded files contains three files: **train2014.tar.gz, val2014.tar.gz, and test2015.tar.gz**, corresponding to the features of the train/val/test images for *VQA-v2*, respectively.

All the image feature files are unzipped and placed in the `data/vqa/feats` folder to form the following tree structure:

```
|-- data
        |-- vqa
        |   |-- feats
        |   |   |-- train2014
        |   |   |   |-- COCO_train2014_...jpg.npz
        |   |   |   |-- ...
        |   |   |-- val2014
        |   |   |   |-- COCO_val2014_...jpg.npz
        |   |   |   |-- ...
        |   |   |-- test2015
        |   |   |   |-- COCO_test2015_...jpg.npz
        |   |   |   |-- ...
```

- QA Annotations

Download all the annotation `json` files for VQA-v2, including the train questions, val questions, test questions, train answers, and val answers.

In addition, we use the VQA samples from the Visual Genome to augment the training samples. We pre-processed these samples by two rules:

1. Select the QA pairs with the corresponding images appear in the MS-COCO *train* and *val* splits;
2. Select the QA pairs with the answer appear in the processed answer list (occurs more than 8 times in whole *VQA-v2* answers).

We provide our processed vg questions and annotations files, you can download them from OneDrive or BaiduYun.

All the QA annotation files are unzipped and placed in the `data/vqa/raw` folder to form the following tree structure:

```
|-- data
        |-- vqa
        |   |-- raw
        |   |   |-- v2_OpenEnded_mscoco_train2014_questions.json
        |   |   |-- v2_OpenEnded_mscoco_val2014_questions.json
        |   |   |-- v2_OpenEnded_mscoco_test2015_questions.json
        |   |   |-- v2_OpenEnded_mscoco_test-dev2015_questions.json
        |   |   |-- v2_mscoco_train2014_annotations.json
        |   |   |-- v2_mscoco_val2014_annotations.json
        |   |   |-- VG_questions.json
        |   |   |-- VG_annotations.json
```

1.2.2 GQA

- Image Features

Download the spatial features and object features for GQA from its official website. **Spatial Features Files** include `gqa_spatial_*.h5` and `gqa_spatial_info.json`. **Object Features Files** include `gqa_objects_*.h5` and `gqa_objects_info.json`.To make the input features consistent with those for VQA-v2, we provide a script to transform `.h5` feature files into multiple `.npz` files, with each file corresponding to one image.

```
$ cd data/gqa

$ unzip spatialFeatures.zip
$ python gqa_feat_preproc.py --mode=spatial --spatial_dir=./spatialFeatures --out_
↪dir=./feats/gqa-grid
$ rm -r spatialFeatures.zip ./spatialFeatures

$ unzip objectFeatures.zip
$ python gqa_feat_preproc.py --mode=object --object_dir=./objectFeatures --out_dir=./
↪feats/gqa-frcn
$ rm -r objectFeatures.zip ./objectFeatures
```

All the processed feature files are placed in the `data/gqa/feats` folder to form the following tree structure:

```
|-- data
        |-- gqa
        |   |-- feats
        |   |   |-- gqa-frcn
        |   |   |   |-- 1.npz
        |   |   |   |-- ...
        |   |   |-- gqa-grid
        |   |   |   |-- 1.npz
        |   |   |   |-- ...
```

- Questions and Scene Graphs

Download all the GQA QA files from the official site, including all the splits needed for training, validation and testing. Download the scene graphs files for `train` and `val` splits from the official site. Download the supporting files from the official site, including the `train` and `val` choices supporting files for the evaluation.

All the question files and scene graph files are unzipped and placed in the `data/gqa/raw` folder to form the following tree structure:

```
|-- data
        |-- gqa
        |   |-- raw
        |   |   |-- questions1.2
        |   |   |   |-- train_all_questions
        |   |   |   |   |-- train_all_questions_0.json
        |   |   |   |   |-- ...
        |   |   |   |   |-- train_all_questions_9.json
        |   |   |   |-- train_balanced_questions.json
        |   |   |   |-- val_all_questions.json
        |   |   |   |-- val_balanced_questions.json
        |   |   |   |-- testdev_all_questions.json
        |   |   |   |-- testdev_balanced_questions.json
        |   |   |   |-- test_all_questions.json
        |   |   |   |-- test_balanced_questions.json
        |   |   |   |-- challenge_all_questions.json
        |   |   |   |-- challenge_balanced_questions.json
        |   |   |   |-- submission_all_questions.json
        |   |   |-- eval
        |   |   |   |-- train_choices
```

(continues on next page)

(continued from previous page)

```
        |   |   |   |   |-- train_all_questions_0.json
        |   |   |   |   |-- ...
        |   |   |   |   |-- train_all_questions_9.json
        |   |   |   |-- val_choices.json
        |   |   |-- sceneGraphs
        |   |   |   |-- train_sceneGraphs.json
        |   |   |   |-- val_sceneGraphs.json
```

1.2.3 CLEVR

- Images, Questions and Scene Graphs

Download all the CLEVR v1.0 from the official site, including all the splits needed for training, validation and testing.

All the image files, question files and scene graph files are unzipped and placed in the `data/clevr/raw` folder to form the following tree structure:

```
|-- data
        |-- clevr
        |   |-- raw
        |   |   |-- images
        |   |   |   |-- train
        |   |   |   |   |-- CLEVR_train_000000.json
        |   |   |   |   |-- ...
        |   |   |   |   |-- CLEVR_train_069999.json
        |   |   |   |-- val
        |   |   |   |   |-- CLEVR_val_000000.json
        |   |   |   |   |-- ...
        |   |   |   |   |-- CLEVR_val_014999.json
        |   |   |   |-- test
        |   |   |   |   |-- CLEVR_test_000000.json
        |   |   |   |   |-- ...
        |   |   |   |   |-- CLEVR_test_014999.json
        |   |   |-- questions
        |   |   |   |-- CLEVR_train_questions.json
        |   |   |   |-- CLEVR_val_questions.json
        |   |   |   |-- CLEVR_test_questions.json
        |   |   |-- scenes
        |   |   |   |-- CLEVR_train_scenes.json
        |   |   |   |-- CLEVR_val_scenes.json
```

- Image Features

To make the input features consistent with those for VQA-v2, we provide a script to extract image features using a pre-trained ResNet-101 model like most previous works did and generate `.h5` files, with each file corresponding to one image.

```
$ cd data/clevr

$ python clevr_extract_feat.py --mode=all --gpu=0
```

All the processed feature files are placed in the `data/clevr/feats` folder to form the following tree structure:

```
|-- data
        |-- clevr
        |   |-- feats
```

(continues on next page)

(continued from previous page)

```
        |   |   |-- train
        |   |   |   |-- 1.npz
        |   |   |   |-- ...
        |   |   |-- val
        |   |   |   |-- 1.npz
        |   |   |   |-- ...
        |   |   |-- test
        |   |   |   |-- 1.npz
        |   |   |   |-- ...
```

CHAPTER 2

Getting Started

This page provides basic tutorials about the usage of mmdetection. For installation instructions, please see Installation.

2.1 Training

The following script will start training a `mcan_small` model on the `VQA-v2` dataset:

```
$ python3 run.py --RUN='train' --MODEL='mcan_small' --DATASET='vqa'
```

- `--RUN={'train','val','test'}` to set the mode to be executed.
- `--MODEL=str`, e.g., to assign the model to be executed.
- `--DATASET={'vqa','gqa','clevr'}` to choose the dataset to be executed.

All checkpoint files will be saved to:

```
ckpts/ckpt_<VERSION>/epoch<EPOCH_NUMBER>.pkl
```

and the training log file will be placed at:

```
results/log/log_run_<VERSION>.txt
```

To add

- `--VERSION=str`, e.g., `--VERSION='v1'` to assign a name for your this model.
- `--GPU=str`, e.g., `--GPU='2'` to train the model on specified GPU device.
- `--SEED=int`, e.g., `--SEED=123` to use a fixed seed to initialize the model, which obtains exactly the same model. Unset it results in random seeds.
- `--NW=int`, e.g., `--NW=8` to accelerate I/O speed.
- `--SPLIT=str` to set the training sets as you want. Setting `--SPLIT='train'` will trigger the evaluation script to run the validation score after every epoch automatically.

- `--RESUME=True` to start training with saved checkpoint parameters. In this stage, you should assign the checkpoint version`--CKPT_V=str` and the resumed epoch number `CKPT_E=int`.
- `--MAX_EPOCH=int` to stop training at a specified epoch number.

If you want to resume training from an existing checkpoint, you can use the following script:

```
$ python3 run.py --RUN='train' --MODEL='mcan_small' --DATASET='vqa' --CKPT_V=str --
↪CKPT_E=int
```

where the args `CKPT_V` and `CKPT_E` must be specified, corresponding to the version and epoch number of the loaded model.

2.1.1 Multi-GPU Training and Gradient Accumulation

We recommend to use the GPU with at least 8 GB memory, but if you don't have such device, we provide two solutions to deal with it:

- *Multi-GPU Training*:

  If you want to accelerate training or train the model on a device with limited GPU memory, you can use more than one GPUs:

  Add `--GPU='0, 1, 2, 3...'`

  The batch size on each GPU will be adjusted to `BATCH_SIZE`/#GPUs automatically.

- *Gradient Accumulation*:

  If you only have one GPU less than 8GB, an alternative strategy is provided to use the gradient accumulation during training:

  Add `--ACCU=n`

  This makes the optimizer accumulate gradients for`n` small batches and update the model weights at once. It is worth noting that `BATCH_SIZE` must be divided by `n` to run this mode correctly.

2.2 Validation and Testing

Warning: The args `--MODEL` and `--DATASET` should be set to the same values as those in the training stage.

2.2.1 Validation on Local Machine

Offline evaluation on local machine only support the evaluations on the *val* split. If you want to evaluate the *test* split, please see [Evaluation on online server](#Evaluation on online server).

There are two ways to start:

(Recommend)

```
$ python3 run.py --RUN='val' --MODEL=str --DATASET='{vqa,gqa,clevr}' --CKPT_V=str --
↪CKPT_E=int
```

or use the absolute path instead:

```
$ python3 run.py --RUN='val' --MODEL=str --DATASET='{vqa,gqa,clevr}' --CKPT_PATH=str
```

- For VQA-v2, the results on *val* split

2.2.2 Testing on Online Server

All the evaluations on the test split of VQA-v2, GQA and CLEVR benchmarks can be achieved by using

```
$ python3 run.py --RUN='test' --MODEL=str --DATASET='{vqa,gqa,clevr}' --CKPT_V=str --
↪CKPT_E=int
```

Result file are saved at: `results/result_test/result_run_<CKPT_V>_<CKPT_E>.json`

- For VQA-v2, the result file is uploaded the VQA challenge website to evaluate the scores on *test-dev* or *test-std* split.
- For GQA, the result file is uploaded to the GQA Challenge website to evaluate the scores on *test* or *test-dev* split.
- For CLEVR, the result file can be evaluated via sending an email to the author Justin Johnson with attaching this file, and he will reply the scores via email too.

CHAPTER 3

Benchmark and Model Zoo

3.1 Environment

We use the following environment to run all the experiments in this page.

- Python 3.6
- PyTorch 0.4.1
- CUDA 9.0.176
- CUDNN 7.0.4

3.2 VQA-v2

We provide three groups of results (including the accuracies of *Overall*, *Yes/No*, *Number* and *Other*) for each model on VQA-v2 using different training schemes as follows. We provide pre-trained models for the latter two schemes.

- **Train -> Val**: trained on the `train` split and evaluated on the `val` split.
- **Train+val -> Test-dev**: trained on the `train+val` splits and evaluated on the `test-dev` split.
- **Train+val+vg -> Test-dev**: trained on the `train+val+vg` splits and evaluated on the `test-dev` split.

Note that for one model, the used base learning rate in the two schemes may be different, you should modify this setting in the config file to reproduce the results.

3.2.1 Train -> Val

3.2.2 Train+val -> Test-dev

3.2.3 Train+val+vg -> Test-dev

3.3 GQA

We provide a group of results (including *Accuracy*, *Binary*, *Open*, *Validity*, *Plausibility*, *Consistency*, *Distribution*) for each model on GQA as follows.

- **Train+val -> Test-dev**: trained on the `train(balance) + val(balance)` splits and evaluated on the `test-dev(balance)` split.

The results shown in the following are obtained from the online server. Note that the offline Test-dev result is evaluated by the provided offical script, which results in slight difference compared to the online result due to some unknown reasons.

3.3.1 Train+val -> Test-dev

3.4 CLEVR

We provide a group of results (including *Overall*, *Count*, *Exist*, *Compare Numbers*, *Query Attribute*, *Compare Attribute*) for each model on CLEVR as follows.

- **Train -> Val**: trained on the `train` split and evaluated on the `val` split.

3.4.1 Train -> Val

CHAPTER 4

Adding a custom VQA model

This is a tutorial on how to add a custom VQA model into OpenVQA. Follow the steps below, you will obtain a model that can run across VQA/GQA/CLEVR datasets.

4.1 1. Preliminary

All implemented models are placed at `<openvqa>/openvqa/models/`, so the first thing to do is to create a folder there for your VQA model named by `<YOU_MODEL_NAME>`. After that, all your model related files will be placed in the folder `<openvqa>/openvqa/models/<YOU_MODEL_NAME>/`.

4.2 2. Dataset Adapter

Create a python file `<openvqa>/openvqa/models/<YOU_MODEL_NAME>/adapter.py` to bridge your model and different datasets. Different datasets have different input features, thus resulting in different operators to handle the features.

4.2.1 Input

Input features (packed as `feat_dict`) for different datasets.

4.2.2 Output

Customized pre-processed features to be fed into the model.

4.2.3 Adapter Template

```
from openvqa.core.base_dataset import BaseAdapter
class Adapter(BaseAdapter):
    def __init__(self, __C):
        super(Adapter, self).__init__(__C)
        self.__C = __C

    def vqa_init(self, __C):
            # Your Implementation

    def gqa_init(self, __C):
            # Your Implementation

    def clevr_init(self, __C):
            # Your Implementation

    def vqa_forward(self, feat_dict):
            # Your Implementation

    def gqa_forward(self, feat_dict):
            # Your Implementation

    def clevr_forward(self, feat_dict):
            # Your Implementation
```

Each dataset-specific initiation function `def <dataset>_init(self, __C)` corresponds to one feed-forward function `def <dataset>_forward(self, feat_dict)`, your implementations should follow the principles `torch.nn.Module.__init__()` and `torch.nn.Module.forward()`, respectively.

The variable `feat_dict` consists of the input feature names for the datasets, which corresponds to the definitions in `<openvqa>/openvqa/core/base_cfg.py`

```
vqa:{
        'FRCN_FEAT': buttom-up features -> [batchsize, num_bbox, 2048],
        'BBOX_FEAT': bbox coordinates -> [batchsize, num_bbox, 5],
}
gqa:{
        'FRCN_FEAT': official buttom-up features -> [batchsize, num_bbox, 2048],
        'BBOX_FEAT': official bbox coordinates -> [batchsize, num_bbox, 5],
        'GRID_FEAT': official resnet grid features -> [batchsize, num_grid, 2048],
}
clevr:{
        'GRID_FEAT': resnet grid features -> [batchsize, num_grid, 1024],
}
```

More detailed examples can be referred to the adapter for the MCAN model.

4.3 3. Definition of model hyper-parameters

Create a python file named `<openvqa>/openvqa/models/<YOUR MODEL NAME>/model_cfgs.py`

4.3.1 Configuration Template

```
from openvqa.core.base_cfgs import BaseCfgs
class Cfgs(BaseCfgs):
    def __init__(self):
        super(Cfgs, self).__init__()
        # Your Implementation
```

Only the variable you defined here can be used in the network. The variable value can be override in the running configuration file described later.

4.3.2 Example

```
# model_cfgs.py
from openvqa.core.base_cfgs import BaseCfgs
class Cfgs(BaseCfgs):
    def __init__(self):
        super(Cfgs, self).__init__()
        self.LAYER = 6
```

```
# net.py
class Net(nn.Module):
    def __init__(self, __C, pretrained_emb, token_size, answer_size):
        super(Net, self).__init__()
        self.__C = __C

        print(__C.LAYER)
```

```
Output: 6
```

4.4 4. Main body

Create a python file for the main body of the model as `<openvqa>/openvqa/models/<YOUR MODEL NAME>/net.py`. Note that the filename must be `net.py` since this filename will be invoked by the running script. Except the file, other auxiliary model files invoked by `net.py` can be named arbitrarily.

When implementation, you should pay attention to the following restrictions:

- The main module should be named `Net`, i.e., `class Net(nn.Module):`
- The `init` function has three input variables: *pretrained_emb* corresponds to the GloVe embedding features for the question; *token_size* corresponds to the number of all dataset words; *answer_size* corresponds to the number of classes for prediction.
- The `forward` function has four input variables: *frcn_feat*, *grid_feat*, *bbox_feat*, *ques_ix*.
- In the `init` function, you should initialize the `Adapter` which you've already defined above. In the `forward` function, you should feed *frcn_feat*, *grid_feat*, *bbox_feat* into the `Adapter` to obtain the processed image features.
- Return a prediction tensor of size [batch_size, answer_size]. Note that no activation function like `sigmoid` or `softmax` is appended on the prediction. The activation has been designed for the prediction in the loss function outside.

4.4.1 Model Template

```
import torch.nn as nn
from openvqa.models.mcan.adapter import Adapter
class Net(nn.Module):
    def __init__(self, __C, pretrained_emb, token_size, answer_size):
            super(Net, self).__init__()
            self.__C = __C
                self.adapter = Adapter(__C)

          def forward(self, frcn_feat, grid_feat, bbox_feat, ques_ix):
                  img_feat = self.adapter(frcn_feat, grid_feat, bbox_feat)
                  # model implementation
              ...

                return pred
```

4.5 5. Declaration of running configurations

Create a `yml` file at`<openvqa>/configs/<dataset>/<YOUR_CONFIG_NAME>.yml` and define your hyper-parameters here. We suggest that `<YOUR_CONFIG_NAME>`= `<YOUR_MODEL_NAME>`. If you have the requirement to have one base model support the running scripts for different variants. (e.g., MFB and MFH), you can have different yml files (e.g., `mfb.yml` and `mfh.yml`) and use the `MODEL_USE` param in the yml file to specify the actual used model (i.e., mfb).

4.5.1 Example:

```
MODEL_USE: <YOUR MODEL NAME>  # Must be defined
LAYER: 6
LOSS_FUNC: bce
LOSS_REDUCTION: sum
```

Finally, to register the added model to the running script, you can modify `<openvqa/run.py>` by adding your `<YOUR_CONFIG_NAME>` into the arguments for models here.

By doing all the steps above, you are able to use `--MODEL=<YOUR_CONFIG_NAME>` to train/val/test your model like other provided models. For more information about the usage of the running script, please refer to the Getting Started page.

CHAPTER 5

Contributing to OpenVQA

All kinds of contributions are welcome, including but not limited to the following.

- Fixes (typo, bugs)
- New features and components

5.1 Workflow

1. fork and pull the latest version of OpenVQA
2. checkout a new branch (do not use master branch for PRs)
3. commit your changes
4. create a PR

5.2 Code style

5.2.1 Python

We adopt PEP8 as the preferred code style. We use flake8 as the linter and yapf as the formatter. Please upgrade to the latest yapf (>=0.27.0) and refer to the configuration.

> Before you create a PR, make sure that your code lints and is formatted by yapf.

5.2.2 C++ and CUDA

We follow the Google C++ Style Guide.

This repo is currently maintained by Zhou Yu (@yuzcccc) and Yuhao Cui (@cuiyuhao1996).

This version of the documentation was built on Sep 03, 2021.