DSAC Code Documentation

November 2, 2017

This document explains the high level concepts and setup of our DSAC code for camera localization. You find an explanation of the method and theory in our CVPR 17 paper. Please cite this paper, if you use this code in your own work:

E. Brachmann, A. Krull, S. Nowozin, J. Shotton, F. Michel, S. Gumhold, C. Rother, "DSAC – Differentiable RANSAC for Camera Localization", CVPR 2017

Note: Beginning of August 2017, we updated the public version of the code to contain a fix in the pose evaluation metric, and to utilize a more stable variant of the PnP algorithm. These changes result in improved numbers compared to the original version of the paper. The improved numbers and a more detailed explanation can be found in the current version of the paper on arXiv. The current version of the code contains both fixes, and also the pre-trained models have been updated, accordingly.

Compiling the code creates six programs:

- `train_obj`: Trains a scene coordinate regression CNN given RGB images and scene coordinate ground truth (see Sec. 4.1. in our paper).
- `train_score`: Trains a score regression CNN given RGB images, a scene coordinate regression CNN and ground truth poses (see Sec. 4.1. in our paper).
- `train_ransac`: Trains the whole camera localization pipeline end-to-end using DSAC (see Sec. 4.2. in our paper).
- `train_ransac_softam`: Trains the whole camera localization pipeline end-to-end using soft argmax instead of DSAC (see Sec. 4.2. in our paper).
- `test_ransac`: Runs the camera localization pipeline on test images with DSAC or RANSAC.
- `test_ransac_softam`: Runs the camera localization pipeline on test images with the soft argmax option instead of DSAC/RANSAC.

Compiling the code will require: OpenCV 2.4, PNG++, Torch and cuDNN. We compiled the code using Ubuntu 14.04 and GCC 4.8.4.

In the following, we will first give more details about the general setup and the individual programs. Then, we describe how to run the code specifically on the 7-Scenes data set.

In case you have questions regarding the code, feel free to contact the first author of the associated paper.

1 General Setup

The DSAC source code is a combination of a C++ framework and multiple Torch scripts. All data access, geometric operations and accuracy evaluation is performed in C++. The Torch scripts are called from C++ to grant access to the CNNs used in the pose estimation pipeline. We use two CNNs, one for regressing scene coordinates given RGB image patches and another one regressing hypothesis scores given re-projection error images. Hence, there are two separate Torch scripts, each one responsible for one of the CNNs. The Torch scripts are contained in the sub-folder `core/lua`. The script variants ending with `_softam` are used in the soft argmax variant of the code. They offer the exact same functionality as the standard scripts but store their respective CNNs under a different file name.

1.1 Setting Parameters

All C++ programs share a set of parameters which the user can specify, see below for a complete list. There are two ways of setting the parameters. The first and direct way is to append it to the program call in the command line. Every parameter has a few letter abbreviation (case sensitive, see below) which can be given to the command line starting with a dash and followed by a space and the value to set, e.g. `-ih 480 -iw 640`. Because there might be a large number of parameters to set, there is also the second way which is providing a `default.config` file. This file should contain one parameter per line. More specifically, each line should start with the parameter abbreviation (without a leading dash) followed by a space and the value to set. The config file can also include comment lines which start with the # symbol. All programs will first look for the `default.config` file which will be parsed at the very beginning of each run. After that, command line parameters will be processed (overwriting parameters from the `default.config`). Our 7-Scenes data package contains an example `default.config` file (see Sec. 3).

Parameters related to the data:

iw width of input images (px)

ih height of input images (px)

fl focal length of the RGB camera

xs x position of the principal point of the RGB camera

ys y position of the principal point of the RGB camera

rd 1 if RGB and depth channels are **not** registered (see Sec. 1.4)

sfl focal length of the depth camera (only if `-rd 1`)

rxs x position of the principal point of the depth camera (only if `-rd 1`)

rys y position of the principal point of the depth camera (only if `-rd 1`)

oscript Torch script file for learning scene coordinate regression

sscript Torch script file for learning hypothesis score regression

omodel file storing the scene coordinate regression CNN

smodel file storing the score regression CNN

Parameters related to pose estimation:

rdraw draw a hypothesis randomly (`-rdraw 1`), i.e. DSAC, or take the one with the largest score (`-rdraw 0`), i.e. RANSAC

rT2D re-projection error threshold (in px) for measuring inliers in the pose estimation pipeline

rT3D inlier threshold (in mm) for evaluation of the intermediate scene coordinate prediction

rI initial number of pose hypotheses drawn per frame

rRI number of refinement iterations

rB max. inlier count for refinement

rSS ratio of pixels for which gradients are calculated during refinement

Parameters related to learning: Parameters which concern the learning of the scene coordinate regression CNN and the score regression CNN (e.g. learning rate or file names to store the CNNs) are defined directly in the corresponding Torch scripts (see `core/lua`).

There are also some parameters defined in the respective main *.cpp file of each program, such as the total number of training iterations or training batch sizes. Also, all programs related to training have a validation compile flag defined in their respective main *.cpp file. If this flag is set to 1, validation passes will be performed in fixed intervals during training. In this case, a validation set has to be available (see Sec. 3).

1.2 End-to-End Training

As stated in the paper, end-to-end training (i.e. calling `train_ransac`) needs a good initialization in terms of the CNNs used. Therefore, component-wise pre-training (i.e. calling `train_obj` followed by `train_score`) should be executed first. An example order of executing the programs is given in Sec. 3 for the 7-Scenes data set.

1.3 Scene Coordinate Ground Truth

In order to pre-train the scene coordinate regression CNN (i.e. calling `train_obj`), scene coordinate ground truth has to be available for each training frame. Our code generates scene coordinate ground truth from a depth frame, the ground truth pose and camera calibration parameters. In general, scene coordinate ground truth can also be rendered using a 3D model of the scene (which is a by-product of most methods generating pose ground truth, like sparse structure-from-motion). In this case, we advise to simply render depth frames and let the code generate scene coordinate ground truth from these. Alternatively, depth frames could also be generated using depth-from-stereo with arbitrary pairs of scene frames.

1.4 Registration of Depth and RGB

When using depth to calculate scene coordinate ground truth, depth and RGB frames have to be registered which is not always the case (e.g. for the 7-Scenes data set). Our code supports mapping depth to RGB if calibration parameters for both the RGB sensor (`-fl -xs -ys`) and the depth sensor (`-sfl -rxs -rys`) as well as the relative rotation/translation between the RGB and depth sensor are known. The latter transformation is assumed to be contained in a file called `sensorTrans.dat`. Our 7-Scenes data package contains an example `sensorTrans.dat` file (see Sec. 3). If such a registration should be performed, set `-rd 1`. If depth and RGB are already registered, set `-rd 0`.

2 Programs

In the following, we describe input and output of the individual programs created by compiling the code. An example order of executing the programs is given in Sec. 3 for the 7-Scenes data set.

2.1 `train_obj`

The program `train_obj` pre-trains a scene coordinate regression CNN by minimizing the L1 distance between the predicted coordinate and the ground truth coordinate. The program needs access to a Torch script which constructs and grants access to a scene coordinate regression CNN (default: `train_obj.lua`). The program will store the current state of the CNN in a fixed interval of parameter updates. This interval and the file name is defined in the Torch script. The program creates a `training_loss` txt file which contains per line the training iteration number followed by the training loss at that iteration (mean over batch). In case the validation compile flag (see `train_obj.cpp`) is set to 1, a `validation_loss` txt file will be created, additionally, which contains per line the training iteration number followed by the mean loss over the validation set at that iteration, and finally the ratio of inlier predictions on the validation set (i.e. ratio of coordinate prediction with a distance smaller than `rT3D` compared to ground truth).

2.2 `train_score`

The program `train_score` pre-trains a score regression CNN by minimizing the difference between the predicted score and a ground truth score. It requires ground truth poses to calculate ground truth scores, and a pre-trained scene coordinate regression CNN (see above) to calculate re-projection error images as input for training. The program also needs access to a Torch script which constructs and grants access to a score regression CNN (default: `train_score.lua`). The program will store the current state of the CNN in a fixed interval of parameter updates. This interval and the file name is defined in the Torch script. The program creates a `score_training_loss` txt file which contains per line the training iteration number followed by the training loss at that iteration (mean over batch). In case the validation compile flag (see `train_score.cpp`) is set to 1, a `score_validation_loss` txt file will be created, additionally, which contains per line the training iteration number followed by the mean loss over the

validation set at that iteration, and finally the ratio of validation batches where the pose with the highest score was also a correct pose (i.e. with an pose error smaller than 5cm 5° compared to the ground truth pose).

2.3 `train_ransac` and `train_ransac_softam`

The program `train_ransac` loads a pre-trained scene coordinate regression CNN and a pre-trained score regression CNN and refines them by training the complete camera localization pipeline end-to-end using the DSAC strategy. The program optimizes the expectation of the pose loss. A variant, namely `train_ransac_softam`, is also available which uses the soft argmax strategy instead of DSAC. The program needs access to two Torch scripts to load and access the two CNNs (default: `train_obj.lua` and `train_score.lua`). The CNNs to load can be specified via `-omodel` resp. `-smodel`. The program will store the current states of both CNNs in fixed intervals of parameter updates. These intervals and the file names are defined in the Torch scripts. The program also creates a `ransac_training_loss` txt file which contains training statistics per training iteration (see `train_ransac.cpp`, line 395 for a listing). In case the validation compile flag (see `train_ransac.cpp`) is set to 1, a `ransac_validation_loss` txt file will be created additionally, which contains validation statistics per validation round (see `train_ransac.cpp`, line 218 for a listing).

2.4 `test_ransac` and `test_ransac_softam`

The program `test_ransac` loads a scene coordinate regression CNN and a score regression CNN and performs camera localization on a set of test images using RANSAC (`-rdraw 0`) or DSAC `-rdraw 1`. A variant, namely `test_ransac_softam`, is also available which uses the soft argmax strategy instead of RANSAC/DSAC. The program needs access to two Torch scripts to load and access the two CNNs (default: `train_obj.lua` and `train_score.lua`). The CNNs to load can be specified via `-omodel` resp. `-smodel`. The program creates a `ransac_test_loss` txt file which contains statistics of the test run (see `test_ransac.cpp`, line 213 for a listing). Most importantly, the first number is the ratio of test images with correctly estimated poses (i.e. a pose error below 5cm 5°). Furthermore, a `ransac_test_errors` txt file will be created, which contains per line statistics for each test image, most importantly the estimated pose (see `test_ransac.cpp`, line 169 for a complete listing).

3 Experiments On 7-Scenes

We provide a data package for deployment of our code for the 7-Scenes data set, namely `7scenes.zip`. Note that we do not provide the data set itself, but merely the structure outline and associated meta data to reproduce the results from our paper. Unpacked, the following structure unfolds:

```
7scenes
    7scenes_chess
        test
            scene
                rgb_noseg
                depth_noseg
                poses
        training
        validation
        default.config
        sensorTrans.dat
        translation.txt
        MyL1Criterion.lua
        train_obj.lua
        train_obj_softam.lua
        train_score.lua
        train_score_softam.lua
    7scenes_fire
    ...
    metadata
```

Marked in red are folders that should be filled with the associated files from the 7-Scenes data set[1], i.e. RGB frames, depth frames and pose files according to the training/test split. The validation folder can be left empty, unless the compile flag in the training programs is set to 1. In this case, a fraction of the training set should be moved to the validation folder.

Marked in blue are files that are identical for all scenes. Hence, they are implemented as relative symbolic links to global directories. For this to work, the `7scenes` folder should be at the same level as the `core` folder of the code (for access to the Torch scripts). The file `default.config` contains the parameter settings used in our experiments. The file `sensorTrans.dat` is used for mapping depth images to RGB images (its the relative rotation/translation between the RGB and depth sensor). The file `translation.txt` contains a normalization translation applied to all poses of the data set. We found that the EPnP implementation of OpenCV is sensitive to translation. We translated each scene, such that its bounding box is centered at (0, 0, 0). This translation is different for each scene. This file is optional.

[1] https://www.microsoft.com/en-us/research/project/rgb-d-dataset-7-scenes/

3.1 Execute DSAC Code on 7-Scenes

To compile the code and re-run our experiments, first adjust the `CMakeLists.txt` file for your particular library versions and locations (e.g. for LUA and Torch), then do the following (assuming all data is in place):

```
# compile code
cd core
mkdir build
cd build
cmake ..
make

# do experiments (e.g. for chess)
cd ../../7scenes/7scenes_chess

# pre-train scene coordinate regression CNN
../../core/build/train_obj -oscript train_obj.lua

# pre-train score regression CNN
../../core/build/train_score -oscript train_obj.lua
        -omodel obj_model_init.net -sscript train_score.lua

# end-to-end training using DSAC
../../core/build/train_ransac -oscript train_obj.lua
        -omodel obj_model_init.net -sscript train_score.lua
        -smodel score_model_init.net

# end-to-end training using SoftAM
../../core/build/train_ransac_softam -oscript train_obj_softam.lua
        -omodel obj_model_init.net -sscript train_score_softam.lua
        -smodel score_model_init.net

# test pipeline with pre-trained CNNs using RANSAC
../../core/build/test_ransac -oscript train_obj.lua
        -omodel obj_model_init.net -sscript train_score.lua
        -smodel score_model_init.net -rdraw 0

# test pipeline with pre-trained CNNs using DSAC
../../core/build/test_ransac -oscript train_obj.lua
        -omodel obj_model_init.net -sscript train_score.lua
        -smodel score_model_init.net -rdraw 1

# test pipeline with pre-trained CNNs using soft argmax
../../core/build/test_ransac_softam -oscript train_obj.lua
        -omodel obj_model_init.net -sscript train_score.lua
        -smodel score_model_init.net

# test pipeline with CNNs trained end-to-end using DSAC
../../core/build/test_ransac -oscript train_obj.lua
        -omodel obj_model_endtoend.net -sscript train_score.lua
        -smodel score_model_endtoend.net -rdraw 1

# test pipeline with CNNs trained end-to-end using soft argmax
../../core/build/test_ransac_softam -oscript train_obj.lua
        -omodel obj_model_softam_endtoend.net -sscript train_score.lua
        -smodel score_model_softam_endtoend.net
```

3.2 Pre-Trained CNNs

On the DSAC project page (not on Github due to space limitations), we provide a package of CNNs for 7-Scenes, trained with the public version of our code, namely `7scenes_models.zip`. The package has the following structure:

```
7scenes
    7scenes_chess
        obj_model_init.net #created by train_obj
        score_model_init.net #created by train_score
        obj_model_endtoend.net #created by train_ransac
        score_model_endtoend.net #created by train_ransac
        obj_model_softam_endtoend.net #created by train_ransac_softam
        score_model_softam_endtoend.net #created by train_ransac_softam
    7scenes_fire
    ...
```