Retina Segmentation Toolbox (Version 1.0)

Fabian Rathke

April 28, 2014

Contents

1 Description **2**

2 Installation **2**

3 Basic Usage **2**
3.1 Configuration . . . 2
3.2 Training . . . 3
3.3 Prediction . . . 3
3.4 Evaluation . . . 3

4 Parameter Glossary **4**
4.1 Collector (`collector.options`) . . . 4
4.1.1 `width` and `height` . . . 4
4.1.2 `clip` and `clipRange` . . . 4
4.1.3 `preprocessing` . . . 4
4.1.4 `loadRoutineData` and `loadRoutineLabels` . . . 4
4.1.5 `numRegionsPerVolume` and `labelIDs` . . . 4
4.1.6 `BScanRegions` . . . 5
4.1.7 `calcOnGPU` . . . 5
4.1.8 `numPatches`, `patchPosition` and `centerPatches` . . . 5
4.1.9 `columnsShape` and `columnnsPred` . . . 5
4.1.10 `printTiming` and `verbose` . . . 5
4.1.11 `saveAppearanceTerms` . . . 5
4.2 Training . . . 5
4.2.1 `appearanceModel` . . . 5
4.2.2 `glasso` . . . 5
4.3 Prediction . . . 5

5 Cross-Validation **5**

6 Labeling Tool **5**
6.1 Configuration . . . 6
6.2 General Usage . . . 6

1 Description

This software package provides a framework for the probabilistic segmentation of retina OCT-scans (2-D and 3-D). It is mainly written in Matlab and C and supports the usage of the GPU via the free Matlab interface GPUmat.

2 Installation

Installation in Matlab is accomplished by adding the source code path via

```
addpath('/path/to/source/code');
```

either into the respective Matlab script or for a permanent adding into the startup.m, a script called by Matlab during each startup. Some functions were written in C, to speed up computations. To be able use these files, they have to compiled from within Matlab using the *mex* function. There are three C functions:

```
cd collector/helperFunctions/
mex getRaster.c
cd ../../prediction/predVariational
mex optQCC.c
cd ../../training/helperFunctions
mex cglasso.c
```

Furthermore, several functions have be implemented such that they can be either run on the CPU or on a Nvidia GPU. This functionality requires the GPUmat toolbox for Matlab and CUDA. See their web page for installing instructions.

3 Basic Usage

In order to segment an OCT scan, we need to *train* appearance models and a shape prior model. To do this we need *ground truth* in form of manually labeled OCT scans. If you have no ground truth available yet, you can use our labeling tool to create some, see Section 6. After we trained appearance and shape models, we can *predict* segmentations of unseen OCT scans. If ground truth is available for these scans, we can *evaluate* the prediction using various error measures.

The subsequent sections introduce the basic usage of each step described above, using the script *scripts/genericCallerScript.m*. To keep this introduction as simple as possible, we rely mostly on the default values for parameters, which correspond to these values used in our recent publication (Rathke et al., 2014). Section 4 will introduce all important parameters and how to use them.

3.1 Configuration

In this section we explain how to set up the *collector*, which is responsible for fetching patches for training and testing/prediction. Most parameters are specified to their default values inside `setCollectorDefaults`.

```
% set folders that hold the data
folderData = '/path/to/data/';
folderLabels = '/path/to/ground/truth/';
files = dir([folderData '*.mat']); % fetch all files
% set up collector; we use the default parameters
```

```
% use 'help setCollectorDefaults' for an overview about model parameters
parameters = struct();
% options that define how to collect ground truth and drawing patches from the OCT-scans
collector.options.labelIDs = repmat(1,length(files),1);
collector.options.loadRoutineData = 'routineData';
collector.options.loadRoutineLabels = 'LabelsFromLabelingTool';
% set the default values for the remaining parameters
collector.options =
    setCollectorDefaults(collector.options,parameters,files,folderData,folderLabels);
% set the collector functions for training and testing
collectorTrn.name = @collectTrnData; collectorTrn.options = collector.options;
collectorTest.name = @collectTestData; collectorTest.options = collector.options;
```

3.2 Training

We set up the approaches for creating appearance and shape models. It is very easy to replace the appearance models by user-defined approaches, i.e. a random forest or support vector machines. After set-up, we start the training procedure by calling `trainModels`.

```
% set the appearance model; we use the default options for both appearance models and shape
    prior
trainFunc.options.appearanceModel = struct('appearanceModel',@trainGlasso);
trainFunc.options.shapeModel = struct();
% train the models; spare the last 5 files for testing the model performance
model = trainModels(files(1:end-5),collectorTrn,parameters,trainFunc.options);
```

3.3 Prediction

After we trained our segmentation model, we are now able to make predictions for unseen scans. Again using mostly default parameters, we want intermediate and final results to be plotted. We then start the prediction procedure by calling `predVariational`.

```
% setup predictor for appearance terms
testFunc.options.predAppearance = @predGlasso;
testFunc.options.plotting = 1;
testFunc.options.folderPlots = '/folder/to/store/plots/';
% make predictions for the last 5 files in the set
prediction =
    predVariational(files(end-4:end),collectorTest,parameters,model,testFunc.options);
```

3.4 Evaluation

Finally, if ground truth is available, we can evaluate segmentation performance in terms of the signed and unsigned error. User-defined error measures can be defined alternatively. We obtain the final results by calling `signedUnsigned`.

```
results = signedUnsigned(prediction,struct());
```

Table 1: Parameters for the collector module. Stored in the struct `collector.options`.

Variable	Type	Default
width	int	15
height	int	15
clip	boolean	false
clipRange	array	[1 scan-width]
preprocessing	struct	options.preprocessing.patchLevel = {{@projToEigenspace,20}}
loadRoutineData	string	'spectralis'
loadRoutineLabels	string	'LabelsFromLabelingTool'
numRegionsPerVolume	int	1
labelIDs	array	zeros(numFiles,numRegionsPerVolume)
BScanRegions	array	[1 numColumns]
calcOnGPU	boolean	false
numPatches	int	30
patchPosition	string	'middle'
centerPatches	boolean	true
columnsShape	cell-array	columnsShape{i} = [1:2:scan-width]
columnsPred	array	1:2:scan-width
printTimings	boolean	false
verbose	int	1
saveAppearanceTerms	boolean	0

4 Parameter Glossary

4.1 Collector (`collector.options`)

4.1.1 `width` and `height`

The patch size in pixel, `width` has to be an odd number.

4.1.2 `clip` and `clipRange`

Enables to clip B-Scans at the left and right boundary, defined via by `clipRange`.

4.1.3 `preprocessing`

Enables preprocessing of each scan at the patch-level or scan-level. `preprocessing` is a struct itself with fields `patchlevel` and `scanlevel`. Each field consists of a cell array where each entry holds the link to the function as the first value and all following values denote parameters. See the corresponding folders in *collector/preprocessing* for available filter functions and their help functions for the required parameters.

4.1.4 `loadRoutineData` and `loadRoutineLabels`

Specifies the loading routines for B-scans and ground truth. For the former, the user has to add a small routine in *loadData.m*. For the latter, if the labels were established with our labeling tool, one can use the default setting for `loadRoutineLabels`. Otherwise, there also must be added a user-defined routine this time in *loadLabels.m*.

4.1.5 `numRegionsPerVolume` and `labelIDs`

For 3-D scans, one has to set the number of B-scans (= regions) that are used from each volume. The accompanying variable labelIDs holds numerical identifier for choosing `numRegionsPerVolume` scans from each 3-D volume. For example there are 50 B-scans per volume,

and one wants to work with only every 5-th, one sets `numRegionsPerVolume=10` and `labelIDs = repmat(1:5:50,numFiles,1)`.

4.1.6 `BScanRegions`

4.1.7 `calcOnGPU`

The switch activates calculation on the GPU. Requires a Nvidia GPU with CUDA and the Matlab toolbox GPUmat installed.

4.1.8 `numPatches`, `patchPosition` and `centerPatches`

The first variables denotes the number of patches drawn per B-scan and appearance class. The columns are chosen evenly distributed across the scan. The second variables determines from where the patches for layer classes are drawn, from the center of each layer ('middle') or from a random position ('random'). The last variable indicates, whether the mean intensity value is subtracted from each patch.

4.1.9 `columnsShape` and `columnnsPred`

`columnsShape` is an cell-array, with one entry for each used scan in the volume. Each entry is an array that gives the columns which will be used for the shape prior. Intermediate columns are interpolated. This allows for sparse shape prior representations. The second variable `columnsPred` is an array with the columns in one B-Scan for each a prediction will be made. Can only be set once for all scans in a volume.

4.1.10 `printTiming` and `verbose`

Prints timings for several subroutines during prediction. The latter determines the level of printed output, takes values between 0 and 2.

4.1.11 `saveAppearanceTerms`

Returns additional to the segmentation the output of the probabilistic appearance models for all pixels in the scan.

4.2 Training

4.2.1 `appearanceModel`

Holds the handle of the appearance model to be used.

4.2.2 `glasso`

4.3 Prediction

4.3.1 `normalizeDataTerm`

5 Cross-Validation

6 Labeling Tool

Included in our software package is a labeling tool that facilitates the creation of ground truth. Fig. 1 shows a B-Scan with 9 boundaries labeled. We provide a generic configuration

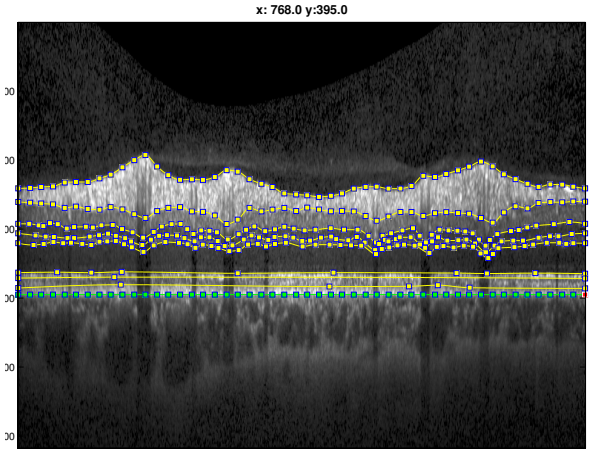

Figure 1: B-Scan with labels provided for 9 boundaries. The scan was flattened along the 9th boundary to ease labeling (press 'r' with the respective boundary selected).

script and the most important options in the next to sections.

6.1 Configuration

```
% the ID of the B-scan (for Spectralis scans either 0 for 2-D or {0,...,61} for volumes)
collector.options.labelID = ID;
collector.options.loadRoutineData = 'dataRoutine';
collector.options.folder_data = '/path/to/oct/scans/';
% if restore == 1 and we have already labeled data with the label tool
collector.options.saveDir = '/path/to/save/labels/';
% if restore == 1, we have not labeled anything yet but there are labels provided with the
    dataset
collector.options.loadRoutineLabels = 'loadRoutine';
collector.options.folder_labels = '/path/to/initial/labels';
% we want no preprocessing (we could add here a Gaussian filter for example or any
    user-defined function to ease labeling)
collector.options.preprocessing.scanLevel = {};
% using existing labels?
restore = 1;
% we label the B-Scan with ID stored inside 'filename.mat';
labelTool('filename.mat',collector,restore);
```

This script can be found in *scripts/exampleLabelingTool.m*.

6.2 General Usage

After you call the function you will see the scan and labels already provided, in case you choose to restore them. First you can crop the scan in order to cut away parts of the scan that are of no interest, first the vitreous body, then the choroid. You can skip each step by pressing 'c'.

After that you can insert new boundaries by pressing 'n' and then clicking inside the scan at the correct height. You can hide boundary lines by pressing 'b'. You can also hide the label markers by pressing 'ctrl + b'. To ease labeling, you can flatten the B-Scan along a boundary by selecting this boundary (use 'q' and 'w' to switch between boundaries) and pressing 'r'. You can fine-tune each label marker by selecting it first (use 'z' and 'x' to cycle

through all markers within the current boundary) and then using the arrow keys. To delete the selected boundary press 'ctrl + d'.

Different modes control the behavior of a left-click. The standard mode is 'add' ('a'), which adds a new label marker at the selected position. The 'edit' ('e') mode moves the nearest marker to the click position. The 'fuse' ('f') mode is useful, if you want to label boundaries around the optical nerve. New marker are added on top of the nearest marker. Finally, the 'delete' ('d') mode erases the closest marker.

These and further options are documented inside the m-file, i.e. call 'help labelTool'.

References

Rathke, F., Schmidt, S., and Schnörr, C. (2014). Probabilistic intra-retinal layer segmentation in 3-D OCT images using global shape regularization. *Med. Image Anal.* in press.