USER MANUAL

Version 1.0

How To Train Your Nao

CDIO HT 2015

Institute of Technology, Linköping University, ISY

Product functionality

The system has four main applications:

- Object model training
- Object detection
- Follow an object
- Searching for an object
- Searching in combination with following

First of all an object model must be trained. When trained models are obtained the robot is able to detect the objects in its field of view. In the follow mode the robot are able to follow detected objects. Finally, if the object is not visible in the field of view of the robot, the robot is able to search for it in its near environment.

Required Hardware and Software

To run the system the following software is required:

- NAOqi 2.1
- OpenCV 2.4.11
- ROS Indigo
- libsvm 3.20
- liuh_driver
- git

To run the system the following hardware is required:

- Nao Robot compatible with NAOqi 2.1
- External computer with required software
- Router where Nao and computer connects

Installation of Software and System

This CDIO project was mainly developed under Linux Mint 17.2 Rafaela.

Installing ROS

Install ROS Indigo by following their instructions on `http://wiki.ros.org/indigo/Installation`
It is probably easiest to install it under distributions based on Ubuntu since it is officially supported by ROS.
ROS is already installed in the humanoids Laboratory of AIICS of IDA. It may be installed in the rest of the computer labs.

Install NAOqi

To interface with the robot the NAOqi API is required on the computer. Download it from the homepage of the manufacturer of Nao, Aldebaran: `https://community.aldebaran.com/en/resources/software/`
Both the C++ NAOqi and Python NAOqi is required, version 2.1.4 was used in this project. It is not possible to download anything without an account on the website. The files are also available in the humanoids lab, under `/sw/aldebaran/`
Cmake (used by ROS) uses *_DIR environment variable to look for external libraries. In this CDIO project the CMakeLists files will look for NAOqi in the AL_DIR variable, in the humanoid lab this is already set. Otherwise AL_DIR should be set to the path where the C++ SDK is located, and the PYTHONPATH to where the Python SDK is unpacked. Keep in mind to not overwrite the PYTHONPATH, best is to concatenate it with the old value. For a private computer add the following two lines to your bashrc, zshrc or equivalent:

```
 $ export AL_DIR=$HOME/path/to/naoqi-sdk
 $ export PYTHONPATH=$PYTHONPATH:/path/to/pynaoqi
```

This is required so catkin (ROS build tool) can find the C++ API and link toward it, and so that Python code can import the NAOqi API.

Install OpenCV

At the university the users don't have access to install libraries to `/usr/local/share` and other system paths. Instead it is needed to install it to the home folder, therefore start with making a folder in your home called OpenCV. It is not necessary to create this folder when building on a private computer.

```
mkdir opencv
```

Since the cv_bridge package in ROS Indigo didn't work with OpenCV 3.0.0 it is preferable to use OpenCV 2.4.11. Download OpenCV from `http://opencv.org/downloads.html`
Unpack the downloaded file, for example to `/tmp` since it is often faster to compile libraries in the `/tmp` folder and if you are working at the university the home folder don't have enough space to build the OpenCV project.

```
 $ mv Downloads/opencv-2.4.11.zip /tmp
 $ cd /tmp
 $ unzip opencv-2.4.11.zip
```

Make a build folder inside the OpenCV folder you just unpacked and move to that new folder.

```
 $ cd opencv-2.4.11
 $ mkdir build
 $ cd build
```

Generate make files, make it and install it to the home folder:

```
 $ cmake -D CMAKE_BUILD_TYPE=RELEASE -D WITH_OPENCL=OFF
 -D CMAKE_INSTALL_PREFIX=/path/to/your/home-folder/opencv ..
 $ make -j8
 $ make install
```

OpenCV is now installed to the home folder but ROS can't find it without telling it where to look. Just as with NAOqi above add the following row to the bashrc, zshrc file or equivalent. This is not required if OpenCV was installed to `/usr/local/share` or similar.

```
export OpenCV_DIR=$HOME/opencv/share/OpenCV
```

Setup the catkin workspace

Create a folder for the ROS workspace and initilize the catkin workspace. Read more at `http://wiki.ros.org/catkin/Tutorials/create_a_workspace`.

```
$ mkdir -p ~/cdio_project/src
$ cd ~/cdio_project/src
$ catkin_init_workspace
```

Download Nao drivers from LiU Humanoids

Download the liuh_driver created by LiU humanoids soccer team from `https://gitlab.ida.liu.se/liu-humanoids/liuh_driver` and it in the src folder. The AL_DIR path set above is used by the liuh_driver/liuh_driver_module/cmake/FindNAOqi.cmake
Remove the liuh_driver_game and liuh_driver_network packages since we don't need them and these depends on other repos.

```
$ cd ~/cdio_project/src/liuh_driver
$ rm -rf liuh_driver_game, liuh_driver_network
```

Download our software

Finally it is time to download the code developed during the CDIO project. Since the project are versioned under git it is easiest to download the project with git:

```
$ cd ~/cdio_project/src/
$ git clone git@gitlab.ida.liu.se:ricbo818/how-to-train-your-nao.git
```

Note that inside the git repo we have embedded the libsvm 3.20 library, so there is no need for installing this library separately.

Build the project

It is possible to verify everything by compiling the project. It may need several tries before it compiles without any errors, this is because of malformatted CMakeLists files that doesn't specify the dependencies between the different packages. An error about HeadMotion or similar will most probably be solved if you compile once again.
Use catkin_make To build a ROS project. A devel and build folder will be created when making the project. In the devel folder setup.bashrc, setup.zshrc and similar files will be created that should be sourced to make the ROS tools find the packages. Remember to source the setup-file that corresponds to the active shell, i.e. setup.zshrc if z shell is used and setup.bashrc if bash is used.

```
$ cd ~/cdio_project
$ catkin_make
$ source devel/setup.bashrc
```

Each time a C++ file is changed the project should be rebuilt with catkin. It is not necessary to rebuild the project when a python file, launch-file or other xml-file is changed.

Running the system

To be able to run the system the above mentioned hardware and software is required. The software system runs with ROS on an external computer which connects to the robot. Below is a guide on how to start the core system and how to run the robot's four main applications. Nothing is actually

started on the robot's hardware, the program and ROS framework is executed on the computer and connects to the robot though proxies. The robot itself runs NAOqi and receives commands from the computer. It is possible to run ROS on the robot and get faster image streams, but the processing power will decrease significantly.
There is two ways to start ROS nodes, either you start up each node separately with rosrun or you launch a bunch of them with roslaunch and a pre-made launch-file. In this project it is easiest to start everything up with launch-files. When using roslaunch the roscore will start automatically, but when using rosrun you'll first need to start the roscore.

Training Object models

An object model must be trained to be able to run the detection, following and searching procedure. In order to train an object model, object data sets with positive and false images must be obtained. To train a good object model it is needed several hundreds of images where the object is visible, but also images without the object. Create a folder for all test images, and sub-folders for each object. When training for a specific object, all other sub-folders will be used as false images. This means that the folder structure will be something like this:

```
test_images
|--tomato
|--cup
|--background
|--empty
+--human
```

When training the tomato model, the images in the cup and background folders will be used as negative images. It is also a good idea to have an empty folder that never is used for training, but the folder will provide more negative images without any objects at all.
The learn node will always be run separately. The learn node doesn't subscribe to anything and doesn't publish anything. The node will read images from the hard drive, from paths specified from the launch-file. The node can be started with rosrun and parameters telling it where to find the images, but it is easier to repeat training using a launch-file. The launch-file is located in the learn package.
The launch-file contains one node, and a parameter array. The array key is the name of the object and its value is the path to the folder containing the images for this model. All sibling folders will be used as negative images.
The parameter can look like this:

```
<node name="learn" type="learn_node" pkg="cdio_learn">
  <param name="objects/tomato" value="/path/to/test_images/tomato"/>
  <param name="objects/cup" value="/path/to/test_images/cup"/>
</node>
```

Note the prefix `objects/` in the name of the parameter, this is the name of the array that is used in the program to load the images. Don't change this name, just change the keys and the paths.
A good way to specify the path to the test images is using the `$(find ros_package)` notation. The code will be more portable without hard-coded paths. Create for example a ROS package outside of the git repository (because git shouldn't contain binary images) and write the parameter as:

```
<param name="objects/tomato" value="$(find datasets)/tomato"/>
```

The data sets package can be located on an external hard drive to save space in the home-folder, but remember to also source the catkin workspace on the hard drive in the same terminal so ROS can find the path to the package. Also note that the hard drive needs to be ext3 or ext4 to be able to initiate a catkin workspace (because vfat and ntfs don't support symlinks).
Start the learn node with:

```
$ roslaunch cdio_learn learn.launch
```

If object isn't found or is badly detected there are most likely too few or bad training images, try to add more. If the model detects a lot of false positives you can try to add the bad detections as false images in a sibling folder.

Start object detection procedure

To be able to detect an object, a model of the object must first be obtained (see above). In order for the robot to be detect an object it must be in the field of view of the robot.
The detection is started with the detect.launch file in the cdio_detector package. This launch-file will start the camera_driver, the yuv2rgb node and the detection node.
The detector node is similar to the learn node considering how to specify which objects should be used in the detection algorithm. The file specifies the names of the objects (used for example by the voice synthesiser) and the path to the model file. The node also take a method parameter which decide which algorithm should be used on the image. Currently it can be "hog" or "surf", but this can be increased in the future. Note that the SURF implementation in OpenCV is marked as nonfree and can't be used in commercial applications.
The SURF algorithm is using Bag of Words and needs a vocabulary file and a SVM model file. The path for surf is only pointing to the filename without extension. The code will then try to load the vocabulary from the `filename.yml` and the SVM model from `filename.xml`.

```
<node name="detector" type="detector_node" pkg="cdio_detector">
  <param name="method" value="surf" />
  <param name="objects/tomato/hog" value="/path/to/hog_tomat" />
  <param name="objects/tomato/surf" value="/path/to/surf_tomat" />
  <param name="objects/cup/hog" value="/path/to/hog_cup" />
</node>
```

The detector node will only load the models for the method specified at the method parameter, i.e. in the code below it will only load the `surf_tomat.yml` and `surf_tomat.xml`. If method would have been hog it will load the `hog_tomat` and `hog_cup` file, note that these files have no extension. If a object is missing a specified method, for example the cup is missing the surf above it wont be detected when this method is used.

```
$ roslaunch cdio_detector general_detect.launch
```

For the human detector another node should be started. This node can be started with:

```
$ roslaunch cdio_detector human_detect.launch
```

Start search and follow procedure

The coordinate node coordinates the perception nodes and the actuator nodes, i.e. make the robot react to his senses.

The coordinate node has three modes:

1. Follow - It can follow objects, always trying to keep the object in the middle of its sight.
2. Search - It can search for an object when it isn't currently in the field of view
3. Follow and search - It will try to follow the object and if it lost the object it will start to search for it.

In the launch-files you can specify the objects to look for by passing in correct models, just as above. To start the coordinate node to follow an object you write:

```
$ roslaunch cdio_coordinator coordinate.launch mode:=follow
```

To start the searching behaviour:

```
$ roslaunch cdio_coordinator coordinate.launch mode:=search
```

And finally to start both searching and following:

```
$ roslaunch cdio_coordinator coordinate.launch mode:=followsearch
```

Note that you can't start follow and search at the same time and get the same effect as followsearch because all three modes are integrated in the same node.
Also note that the coordinate node will spin up the speak node so the robot will say whether it sees an object or not.