Image Recognition for Analog Gauge Reading

Thomas Jefferson National Laboratory

Dr. Brad Sawatzky

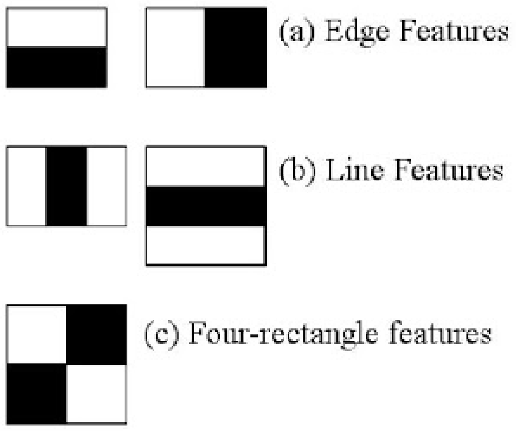
Jonathan Barlow

Motivation

Image recognition is a burgeoning field that has significant applications in a variety of areas such as automatic inspection in manufacturing, medical image analysis, and autonomous navigation. Our goal is to use image recognition software to assist in reading analog gauges at Jefferson Lab. This will allow experimentalists to remotely monitor their analog gauges in a variety of environments. The only components needed are a webcam and Raspberry Pi running the appropriate software and cascade classifiers created during this project.

Haar Feature-Based Cascade Classifier Overview

In 2001, Paul Viola and Michael Jones proposed a new object detection method in their paper titled “Rapid Object Detection using a Boosted Cascade of Simple Features”. In essence, an algorithm is trained to detect distinguishing features of an object by sifting through thousands of positive images (images that contain the object to be detected) and negative images (images not containing the object). Haar features are shown below:



Haar Based Features [1]

Each feature is assigned a single value by subtracting the sum of pixels in the white area from the sum of pixels in the black area. In a 48x48 image there can be hundreds of thousands of unique features, so a machine learning algorithm known as AdaBoost is used to select only the features that best describe the object of interest. Now, the number of features is reduced to a few hundred or thousand. However, it is still time consuming to apply all of these features for each 48x48 window in an image. To reduce computation time further, a concept known as a Cascade of Classifiers is used, which essentially discards regions where it is unnecessary to check for features. Instead of applying all features on a given window, features are grouped into different stages, with each stage containing progressively more features. If a window fails the first stage, discard this region. If a window passes, proceed to the next stage. A window which passes all stages is determined to be a region containing the object of interest.

1 https://docs.opencv.org/master/d7/d8b/tutorial_py_face_detection.html

Procedure

Required Software

I am working on a 2013 MacBook Pro, running OS High Sierra v.10.13.1.

To perform image recognition we used OpenCV, which is a library of programming functions mainly used for real-time computer version. It can be a hassle to configure the download correctly, so for OS X users it's recommended to use a package manager such as Homebrew - which requires an intel CPU, OS X 10.10 or higher, Command Line Tools for Xcode, and a bash shell.

If Command Line Tools isn't already installed, navigate to the terminal and type:

```
xcode-select --install
```

You may also install Xcode directly from the App store, however this will also download items not required. Next, to install Homebrew, paste this line in the terminal:

```
/usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

The script above will explain what it is doing, and not install anything without permission. Here is the documentation for install:

Homebrew Install
https://docs.brew.sh/Installation.html

Check if installed correctly using this line in terminal:

```
brew update
```

Next run these three lines separately in the terminal, this will add Homebrew path in PATH.

```
echo "# Homebrew" >> ~/.bash_profile

echo "export PATH=/usr/local/bin:$PATH" >> ~/.bash_profile

source ~/.bash_profile
```

We can now install OpenCV with Homebrew by executing the following command:

```
brew install opencv
```

This formula installs the newest version of OpenCV 3, and also requires python 2 and 3. If you do not have these brew will automatically install them.

After OpenCV is done downloading, run this command:

```
brew doctor
```

This will then instruct you to update the path of some site-packages so everything works smoothly.

Preparing to train a classifier

Here is the documentation for cascade training and an additional tutorial that I found particularly helpful.

OpenCV Documentation
https://docs.opencv.org/trunk/dc/d88/tutorial_traincascade.html

Naotoshi Seo's Haar Training Tutorial
http://note.sonots.com/SciSoftware/haartraining.html#w0a08ab4

To train an object detector the user must supply many (generally, the more the better) images that contain the object to be detected, these are called positive images. Negative images must also be supplied, which are of anything but the object to be detected. While there is no formal analysis on the optimal positive to negative image count ratio, a common piece of advice is 2:1. Our positive samples were obtained by taking short videos showing the analog gauge pack from various angles. Then a shell command using the 'ffmpeg' tool could be used to generate images from the frames of the video file and store them in a specified directory:

```
for f in *.mp4; do
    ffmpeg -i ../$f ../path/to/store/images/framegrab_%05d.jpg;
done
```

Negative images (~1000) were obtained from a repository using this command [2]:

```
cd path/to/negative/image/directory
wget -nd -r -A "neg-0*.jpg" \
http://haar.thiagohersan.com/haartraining/negatives/
```

A "background collection folder" needs to be made, which is just a text file with the path to each negative image per line. In the terminal:

```
cd path/to/negative/image/directory
ls -l1 *.jpg > bg.txt
```

What we want to do now is crop the positive images so that only the object of interest is in the frame. Instead of going through each picture and cropping by hand, we can employ a batch cropping method to crop many images at once. Because we obtained our positives from a moving video, there will be certain groups of images that have the gauge pack in roughly the same area (when the camera isn't moving a lot). The idea is to find these sections of images, and do a batch crop command. You want to make sure that the area you're cropping will capture the gauge pack in every image you specify to crop. For example:

2 https://mememememememememe.me/post/training-haar-cascades/

```
for f in $(seq -f “%05g” 0 42); do
    convert framegrab_$f.jpg -crop 225x225+20+116
    ../path/to/store/positive/images/cropped_$f.jpg;
done
```

This command would go through all images that are between framegrab_00000.jpg and framegrab_00042.jpg, and crop them to 225x225, starting from 20 pixels right and 116 pixels down from the top left corner of the parent image. They are then stored in my positive image directory, as cropped_(number in sequence).jpg. This can then be repeated for all positive images. Importantly, ensure that the ratio of the crop is consistent across all crops. I chose a simple ratio of 1 (225x225), but this will depend on the shape of your object of interest.

The final tweak needed on our positive images is to resize all cropped images and convert them to grayscale. We want to resize our images to something small like 48x48 (preserving the aspect ratio chosen earlier), to cut down on training time. Even in a small image like 48x48, there can still be one million unique features that the training algorithm will sift through. Similarly, converting to grayscale is computationally economical. These two commands use ImageMagick, download if necessary:

```
magick ‘cropped*.jpg[48x48]’ path/to/store/resized/images_%05d.jpg
```

```
mogrify -type Grayscale /path/to/resized/images/*
```

Next, an info file needs to be created that specifies where the gauge pack is in our 48x48 images. This is why we did all the cropping, because now we can trivially say that the gauge pack starts at the pixel position (0,0) and is 48x48 in area.

```
find /path/to/resized/images -iname “*.jpg” -exec echo \{\} 1 0 0 48 48 \; >
gaugepack.info
```

This creates a file that looks like the following:

```
pos/img00001.jpg 1 0 0 48 48
pos/img00002.jpg 1 0 0 48 48
pos/img00003.jpg 1 0 0 48 48
etc.
```

Generally:

path/to/positive/images [# of objects of interest in image] [x y width height]

The last step before training is essentially a file format conversion function, this takes in our gaugepack.info file and converts it to a vector file format.

```
opencv_createsamples -info gaugepack.info -vec gaugepack.vec -w 48 -h 48
```

Training

Create a directory called data to store the cascade data files. There is only one command to begin training, but there are a lot of options to customize. The OpenCV cascade training documentation explains them all in detail.

```
opencv_traincascade -data /path/to/data/directory -vec gaugepack.vec -bg bg.txt -num-Stages 12 -numPos (# of pos imgs) -numNeg (# of negative imgs) -w 48 -h 48
```

Ensure the width and height parameters (-w -h) are the size of your positive images. Not entirely sure why, but it is recommended to set -numPos and -numNeg to be less than their true value. I did ~150 less and had no issues. -num-Stages is recommended to be between 10 and 15. Anything less will be a weak classifier, and anything more is at risk for overfitting the data. Depending on your computational power and dataset size, training could take from a few hours to multiple days.

<u>Testing</u>

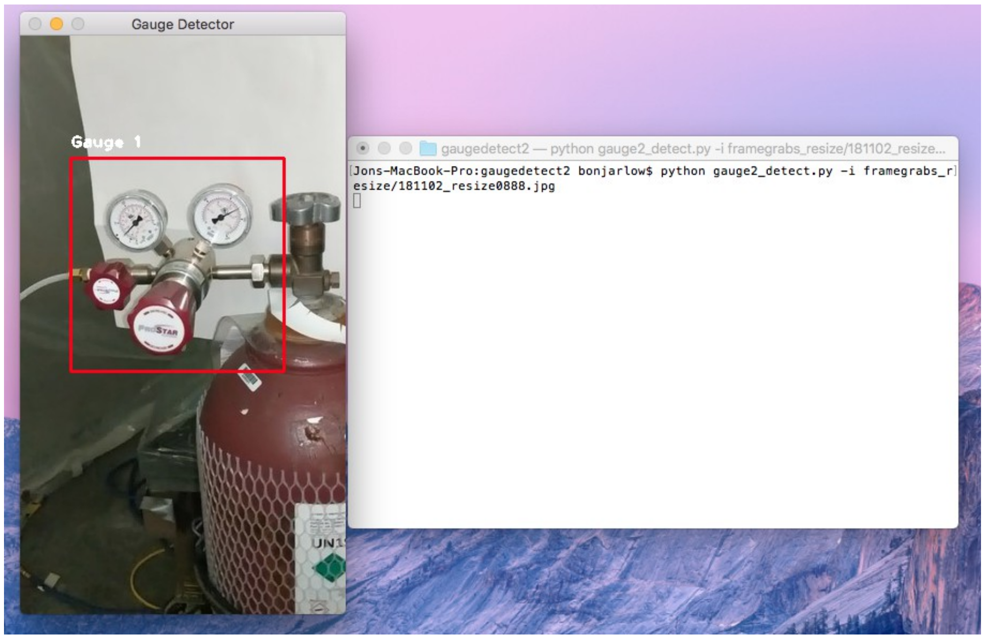
Once the training cascade has completed, a final XML file will be saved in the data directory. We will need a Python script to use this cascade file to visualize our object detector. Navigate to a place in your file system you want to download the necessary files:

```
git clone https://github.com/bonjarlow/fall17.git
```

cd into fall17/gaugedetect2, then in the terminal:

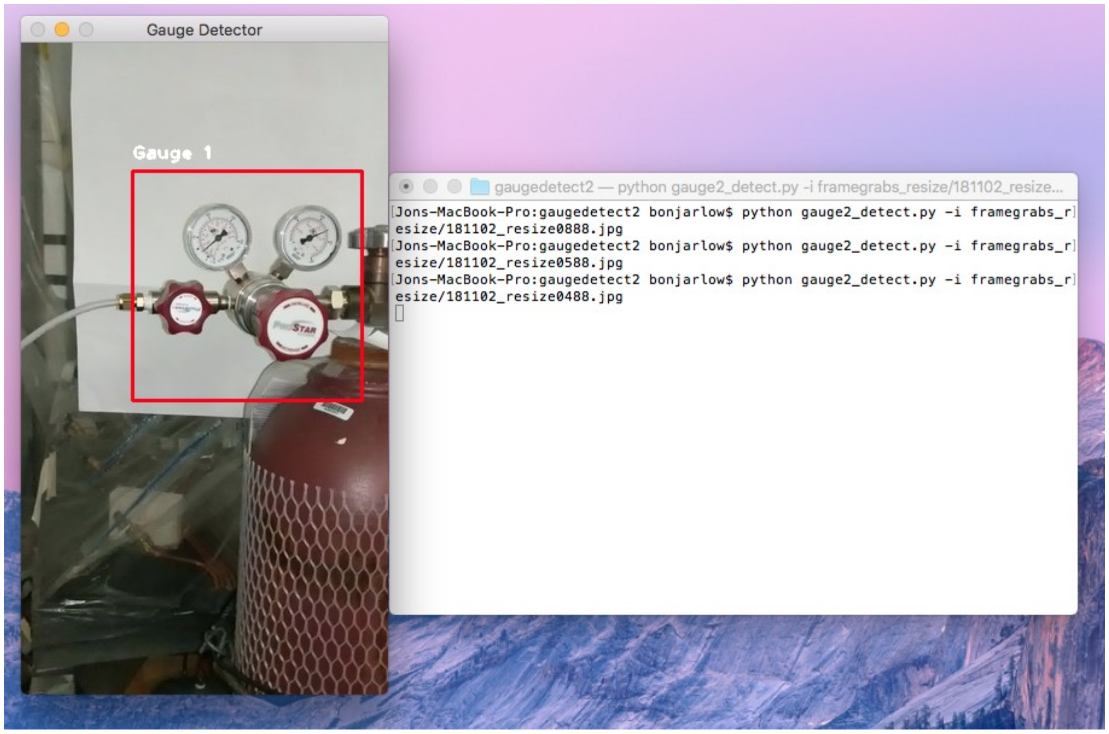
```
python gauge2_detect.py --image path/to/a/positive/image.jpg
```

This will then show a window of the positive image you specified, with (hopefully) a bounding box around it. If the region is not accurate, you may need to go into the Python script and toy with a few parameters. Specifically, scaleFactor, minNeighbors, and minSize. ScaleFactor accounts for object sizes that are different from those in training, minNeighbors is useful for weeding out false detections, and minSize sets the minimum size you can detect. Keep in mind you cannot detect anything smaller than the -w and -h you specified during training (48x48 in our case). Below are some examples of detection:


Gauge Detector
Gauge 1
gaugedetect2 — python gauge2_detect.py -i framegrabs_resize/181102_resize...

```
Jons-MacBook-Pro:gaugedetect2 bonjarlow$ python gauge2_detect.py -i framegrabs_r
esize/181102_resize0888.jpg
```

Gauge Detection, Example 1


Gauge Detector
Gauge 1
gaugedetect2 — python gauge2_detect.py -i framegrabs_resize/181102_resize...

```
Jons-MacBook-Pro:gaugedetect2 bonjarlow$ python gauge2_detect.py -i framegrabs_r
esize/181102_resize0888.jpg
Jons-MacBook-Pro:gaugedetect2 bonjarlow$ python gauge2_detect.py -i framegrabs_r
esize/181102_resize0588.jpg
Jons-MacBook-Pro:gaugedetect2 bonjarlow$ python gauge2_detect.py -i framegrabs_r
esize/181102_resize0488.jpg
```

Gauge Detection, Example 2

Raspberry-Pi and Camera

Finally, we need to have our cascade classifier made above work in conjunction with a Pi Camera. Assuming the Raspberry Pi is configured and the camera module enabled, only a few things remain to be installed:

- OpenCV with Python
- libconfig

The above two can be installed easily using the sudo apt-get command.

- gaugedetect_camera.py
- readGauge.C

The above two can be found in the fall17/raspi_detect folder of my GitHub repo linked earlier. Ensure that these are saved in the same folder on your raspi.

- gauge2_cascade.xml

This last one can be found in the fall17/gaugedetect2/data folder of my GitHub repo. Be sure to compile readGauge.C. Once this is done we are ready to begin detecting through the raspi camera.

The action takes place within gaugedetect_camera.py. This program activates the raspi camera in 60 second intervals, identifies the gauge pack region using the gauge2_cascade.xml file, then sends the image with bounding region information to the readGauge.C program to digitize the dial reading. The output is shown in the terminal. Currently, this output stream is not directed anywhere but the terminal. I have left this to the user :) Also, the photos taken by the raspi camera are saved in gaugepics/ indefinitely, one may want to routinely delete these to conserve storage. Be sure path names are correct within gaugedetect_camera.py!

If everything is ready, in the terminal:

```
$ python gaugedetect_camera.py
```

If you would like to view each image and the bounding box before further processing, un-comment cv2.imshow and cv2.waitKey in gaugedetect_camera.py. You will need to enter a key stroke to progress the camera. For debugging purposes, lower the time.sleep() interval. To quit any process, use ctrl-c on the terminal. If you would like to input images from a folder rather than the camera, use gaugedetect_folder.py instead.