Raspberry Pi for Computer Vision — Table of Contents

You can bring computer vision to the Raspberry Pi, Google Coral, and NVIDIA Jetson Nano.

Computer Vision, **Deep Learning**, and **Internet of Things/Embedded Vision** are three of the *fastest-growing* industries and subjects in computer science — **you will learn how to combine *all three* inside my new book.**

Whether this is the first time you've worked with the embedded devices, or you're a hobbyist who's been working with the embedded systems for years, ***Raspberry Pi for Computer Vision* will enable you to "bring sight" to the RPi, Google Coral, and NVIDIA Jetson Nano.**

Inside this book you'll learn how to:

- **Build practical, real-world computer vision applications** on the Pi
- Create **computer vision and Internet of Things (IoT)** projects and applications with the RPi
- **Optimize your OpenCV code and algorithm**s on the resource constrained Pi
- Perform **Deep Learning on the Raspberry Pi** (including the **Movidius NCS** and **OpenVINO toolkit**)
- Utilize the **Google Coral** and **NVIDIA Jetson Nano** to build embedded computer vision and deep learning applications

Since this book covers a huge amount of content (over 60+ chapters), I've decided to break the book down into ***three volumes*** called **"bundles".** Each bundle *builds on top of the others* and includes *all chapters from the previous bundle.*

You should choose a bundle based on:

1. How in-depth you want to study computer vision and deep learning one embedded devices
2. Which devices you want to utilize (Raspberry Pi, Movidius NCS, Google Coral, and/or Jetson Nano)
3. Your particular budget.

Here is a quick breakdown on what the three bundles cover:

- **Hobbyist Bundle:** A great fit if this is your first time you're working with computer vision, Raspberry Pi, or embedded devices.
- **Hacker Bundle:** Perfect if you want to learn more advanced techniques, including deep learning on embedded devices, working with the Movidius NCS and OpenVINO toolkit, and self-driving car applications. You'll also learn my tips, suggestions, and best practices when applying computer vision on the Raspberry Pi.
- **Complete Bundle:** The complete *Raspberry Pi for Computer Vision* text, including ***19 bonus chapters*** on the **Google Coral** and **NVIDIA Jetson Nano.** You'll have access to every chapter in the book, video tutorials, a hardcopy of the text, and access to my private community and forums for additional help and support.

To see the full list of topics you'll learn inside *Raspberry Pi for Computer Vision*, just keep scrolling…

The *Hobbyist Bundle* includes all of the following chapters…

Contents

The *Hacker Bundle* includes all chapters from the *Hobbyist Bundle,* plus…

Contents

The *Complete Bundle* includes all chapters from the *Hobbyist Bundle* AND the *Hacker Bundle,* plus…

The *Complete Bundle* includes **everything** from the *Hobbyist Bundle* and *Hacker Bundle.*

In additional, it also includes:

- An additional **19 bonus chapters, guides, and tutorials**
- **Video tutorials and walkthroughs** for each chapter
- Access to my **private Raspberry Pi and Computer Vision community and forums**
- A **physical, hardcopy edition of the text** delivered to your doorstep

TensorFlow Lite on the Raspberry Pi

- Learn how to **train your own models using TF Lite**
- **Convert existing models** to TF lite format
- Deploy ***fast, highly optimized*** models to the RPi with TF Lite
- **Perform human pose estimation** using TF Lite

Google Coral and the Raspberry Pi

- Configure your Google Coral USB Accelerator
- Perform **image classification** with the Google Coral
- Create **real-time object detectors** with the Coral
- Train and deploy **your own custom models** using the Coral

NVIDIA Jetson Nano Hands-on Tutorials and Guides

- Configure your Jetson Nano
- Perform **image classification** with the Nano
- **Real-time object detection** with the Jetson Nano
- Train and deploy **your own custom models to the Nano**

Tips, Suggestions, and Best Practices

- When to use the RPi, Google Coral, or NVIDIA Jetson Nano
- Suggestions on when utilize TensorFlow, Keras, or Caffe for training
- How to use ***both*** **the Google Coral USB Accelerator and Jetson Nano** ***together*** for super fast deep learning inference

Note: *All* Complete Bundle *chapters are set to be released in January/February 2019. If you purchase the* Complete Bundle *you will have* ***immediate access*** *to the* Hobbyist Bundle *and* Hacker Bundle *chapters. You'll then receive an email to access the* Complete Bundle *chapters once they are released (and enter your shipping information for the hardcopy edition).*

The Following *“Why the Raspberry Pi?”* chapter is from the *Hobbyist Bundle*…

Chapter 1

Why the Raspberry Pi?

> *"The Raspberry Pi is a low cost, credit-card sized computer that plugs into a computer monitor or TV, and uses a standard keyboard and mouse. It is a capable little device that enables people of all ages to explore computing, and to learn how to program in languages like Scratch and Python."* — Raspberry Pi foundation

Out of all the computer devices in the past decade that have facilitated not only *innovation*, but *education* as well, very few, if any devices have surpassed the Raspberry Pi. And at only $35, this single board computer packs a punch similar to desktop hardware a decade ago.

Since the Raspberry Pi (RPi) was first released back in 2012, the surrounding community has used it for fun, innovative, and practical projects, including:

i. Creating a wireless print server

ii. Utilizing the RPi as a media center

iii. Adding a controller to the RPi and playing retro Atari, NES, SNES, etc. games via a software emulator

iv. Building a stop motion camera

...and the list goes on.

At the core, the Raspberry Pi and associated community has its roots in both:

- **Practicality** — Whatever is built with the RPi should be *useful* in some capacity.
- **Education** — The hacker ethos, at the core, is about *education* and learning something new. When using a RPi, you should be pushing the limits of your understanding and enabling yourself to learn a new technique, method, or algorithm.

I published my first series of Raspberry Pi tutorials on the PyImageSearch blog back in 2015. These tutorials taught PyImageSearch readers how to:

i. Install OpenCV (with Python bindings) on the RPi

ii. Access the Raspberry Pi camera module

iii. Build a home surveillance system capable of detecting motion and sending notifications to the user

These blog posts fit the RPi ethos perfectly. Not only is building a home security application *practical*, but it was also *educational*, both for myself as well as the PyImageSearch community.

Nearly five years later, I feel incredibly lucky and privileged to bring this book to you, keeping the Raspberry Pi ethos close to heart. **Inside this text expect to find highly practical, hands-on computer vision and deep learning projects that will challenge your education and enable to you build real-world applications.**

What are you waiting for? Let's get started!

1.1 Chapter Learning Objectives

In this chapter you will:

- Discover the history of the Raspberry Pi
- Learn how computer vision (CV) can be applied on the RPi
- Briefly review how IoT and Edge Computing is fueling the RPi
- Discover alternative devices for CV and Deep Learning (DL)

1.2 Can We Use the RPi for CV and DL?

In the first part of this chapter we'll briefly review the history of the Raspberry Pi. I'll then discuss how computer vision can be applied to the RPi, followed by how the current trends in Internet of Things (IoT) and Edge Computing applications are helping drive innovation in embedded devices (both at the software and hardware level).

We'll then wrap up by looking at coprocessor devices, such as Intel's Movidius NCS [1] and Google's Coral USB Accelerator [2], and how they are facilitating state-of-the-art deep learning on the RPi.

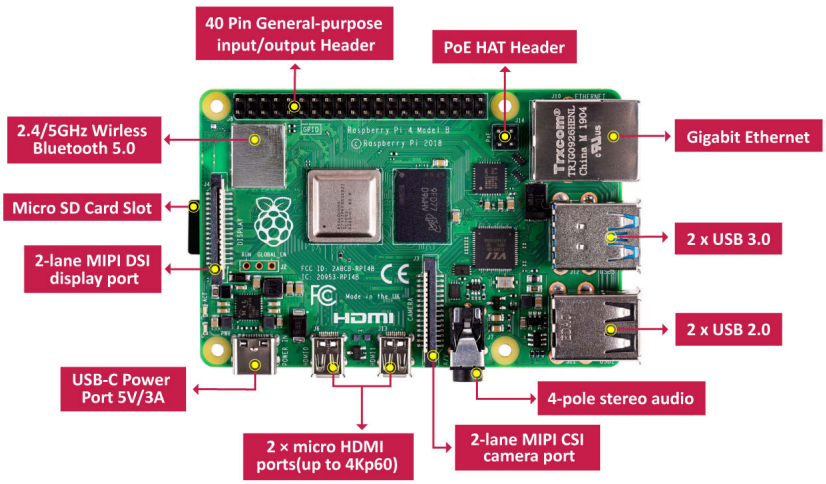

Figure 1.1: The Raspberry Pi 4 comes with a 64-bit 1.5GHz processor and 1-4GB of RAM, all in a device approximately the size of a credit card and under $35. Credit: Seeed Studio [3]

1.2.1 The RPi for CV and DL Applications

The very first Raspberry Pi was released in 2012 with a 700 MHz processor and 512GB of RAM. Specs have improved over the years — the current iteration, the RPi 4B (Figure 1.1), has a 64-bit 1.5GHz quad-core processor and 1-4GB of RAM (depending on the model).

The Raspberry Pi has similar specs to desktop computers from a decade ago, meaning it's still incredibly underpowered, especially compared to our current laptop/desktop devices — **but why should we be interested?**

To start, consider the field of computer vision from a research perspective — image understanding algorithms that were *incredibly computationally expensive* and *only capable of running on high-end machines* ten years ago can now be effectively executed on the RPi.

We can also look at it from a practitioner viewpoint as well — computer vision libraries have matured to the point where they are straightforward to install, simple to use (once you understand them), and *are so highly optimized* that even more recent computer vision algorithms can be run on the RPi.

Effectively, the Raspberry Pi has brought computer vision to embedded devices, whether you are a hobbyist or an experienced practitioner in the field.

And furthermore, we are *not* limited to computer vision algorithms. Using coprocessor de-

vices, such as the Movidius NCS or Google Coral Accelerator, we are now capable of deploying *state-of-the-art* deep neural networks to the RPi as well!

This is an *incredibly* exciting time to be involved in computer vision on embedded devices — the possibility for innovation is nearly endless.

1.2.2 Cheap, Affordable Hardware

Part of what makes the Raspberry Pi so attractive is the relatively cheap, affordable hardware. At the time of this writing, a Raspberry Pi 4 costs $35, making it a minimal investment for both:

i. **Hobbyists** who wish to teach themselves new algorithms and build fun projects.

ii. **Professionals** who are creating products using the RPi hardware.

At only $35, the RPi is well positioned, enabling hobbyists to afford the hardware while providing enough value and computational horsepower for industry professionals to build production-level applications with it.

1.2.3 Computer Vision, Compiled Routines, and Python

It is no secret that computer vision algorithms can be computationally expensive. Typically, when a programmer needs to extract *every last bit of performance* out of a routine, they'll ensure every variable, loop, and construct is optimized, typically by implementing the method in C or C++ (or even dropping down to assembler).

While compiled binaries are undoubtedly fast, the associated code takes significantly longer to write and is often harder to maintain. But on the other hand, languages such as Python, which tends to be easier to write and maintain, often suffer from slower code execution.

Luckily, computer vision, machine learning, and deep learning libraries are now providing compiled packages. Libraries such as OpenCV, scikit-learn, and others:

- Are implemented directly in C/C++ *or* provide compiled Cython optimized functions (Python-like functions with C-like performance).
- Provide Python bindings to interact with the compiled functions.

Effectively, this combination of compiled routines and Python bindings gives us the best of both worlds. We are able to leverage the *speed* of a compiled function while at the same time *maintaining* the ease of coding with Python.

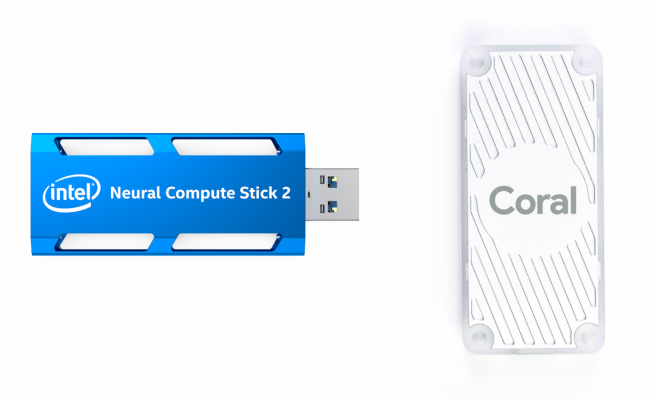

Figure 1.2: **Left:** Intel's Movidius Neural Compute Stick. **Right:** Google Coral USB Accelerator.

1.2.4 The Resurgence of Deep Learning

As we'll discuss in the next chapter, the latest resurgence in deep learning has created an additional interest in embedded device, such as the Raspberry Pi. Deep learning algorithms are *super powerful*, demonstrating unprecedented performance in tasks such as image classification, object detection, and instance segmentation.

The *problem* is that deep learning algorithms are *incredibly* computationally expensive, making them challenging to run on embedded devices.

But just as computer vision libraries are making it easier for CV to be applied to the RPi, the same is true with deep learning. Libraries such as TensorFlow Lite [4] enable deep learning practitioners to train a model on a custom dataset, optimize it, and then deploy it to resource constrained devices as the RPi, obtaining faster inference.

1.2.5 IoT and Edge Computing

The Raspberry Pi is often used for Internet of Things applications. A great example of such a project would be building a remote wildlife detector (which we'll do later in Chapter 13).

Such a system is deployed in the wilderness and is either powered by batteries and/or a solar panel. The camera then captures and processes images of wildlife, useful for approximating species counts and detecting intruders.

Another great example of IoT and edge computing with the RPi comes from Jeff Bass (http://pyimg.co/h8is2), a PyImageSearch reader who uses RPis around his farm to monitor temperature, humidity, sunlight levels, and even detect water meter usage.

Remark. *If you're interested in learning more about how Jeff Bass is using computer vision and RPis around his farm, you can read the full interview on the PyImageSearch blog: http://pyimg.co/sr2gj.*

1.2.6 The Rise of Coprocessor Devices

As mentioned earlier, deep learning algorithms are computationally expensive, which is a *big* problem on the resource constrained Raspberry Pi. In order to run these computationally intense algorithms on the RPi we need additional hardware.

Both the Intel (Movidius NCS) [1] and Google (Coral USB Accelerator) [2] have released what are essentially "USB sticks for deep learning inference" that can be plugged into the RPi (Figure 1.2). We call such devices "coprocessors" as they are designed to augment the capabilities of the primary CPU.

Combined with the optimized libraries from both Google and Intel, we can obtain faster inference on the RPi than using the CPU alone.

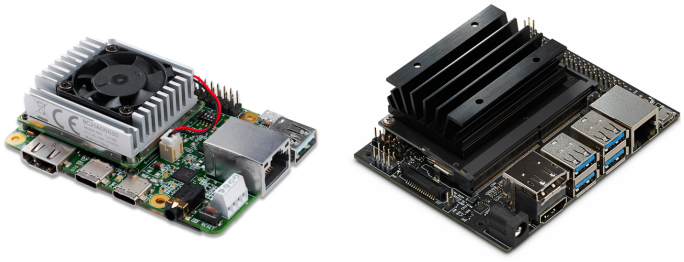

Figure 1.3: **Left:** Google Coral Dev Board. **Right:** NVIDIA Jetson Nano.

1.2.7 Embedded Boards and Devices

Of course, there are situations where the Raspberry Pi itself is not sufficient and additional computational resources are required beyond what coprocessors can achieve. In those cases, you would want to look at Google Coral's Dev Board [5] and NVIDIA's Jetson Nano [6] (Figure 1.3) — these single board computers are similar in size to the RPi but are *much* faster (albeit more expensive).

While the Raspberry Pi is the primary focus of this book, all code is meant to be compatible with minimal (if any) changes on both the Coral and Jetson Nano. I've also included notes in the relevant chapters regarding where I would suggest using an alternative to the RPi.

1.3 Summary

In this chapter we discussed how the Raspberry Pi can be used for computer vision, including how deep learning, IoT, and edge computing are pushing innovation in embedded devices, both at the *hardware* and *software* level.

In the next chapter we'll expand on our knowledge of the RPi within the CV and DL fields, paving the way for you to build computer vision applications on the Raspberry Pi.

This *“Face Tracking with Pan/Tilt Servos”* chapter is from the *Hobbyist Bundle*…

Chapter 18

Face Tracking with Pan/Tilt Servos

Ever since I started working with computer vision, I thought it would be really cool to have a camera track objects. We've accomplished this on the PyImageSearch blog with object detection and tracking algorithms (http://pyimg.co/aiqr5) [71].

But what happens if the object (ex. person, dog, cat, horse, etc.), goes out of the frame?

In that case, usually there is nothing we can do. That is, unless we add mechanics to our camera. There are certainly plenty of pan/tilt security cameras on the market, but usually they are manually controlled by an operator.

Luckily for us, there's a great HAT for us made by Pimoroni which makes *automatic* pan/tilt tracking possible using the Pi Camera. In this chapter we're going to use the Raspberry Pi pan/tilt servo HAT and PIDs to track a moving target using servo mechanics.

18.1 Chapter Learning Objectives

In this chapter, we'll apply and reinforce our knowledge of PIDs from Chapter 17 to track objects. We will learn about the following concepts to accomplish our goal:

i. Multi-processing

ii. Process-safe variables

iii. The Haar Cascade face detector

iv. Tuning our PID gain constants

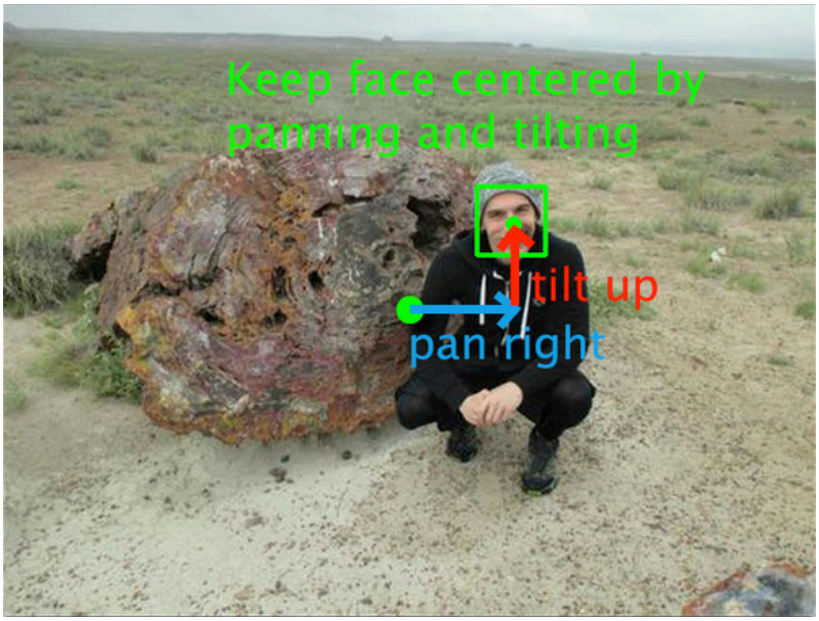

Figure 18.1: The goal of pan/tilt face tracking is to use mechanical servo actuators to follow a moving face in a scene while keeping the face centered in the view of the camera. The following steps take place: (1) the face detector localizes the face, (2) the PID process accepts an error between where the face is and where it should be, (3) the PID update method calculates a new angle to send to the servo, (4) rinse and repeat. These steps are performed by two independent processes for both panning and tilting.

18.2 Pan/tilt Face Tracking

In the first part of this chapter, we'll discuss the hardware requirements. Then we'll briefly review the concept of a PID. From there, we'll dive into our project structure. Our object center algorithm will find the object we are tracking. For this example, it is a face, but don't limit your imagination to tracking only faces.

We'll then walk through the script while coding each of our processes. We have four:

i. **Object center** - Finds the face

ii. **PID A** - Panning

iii. **PID B** - Tilting

iv. **Set servos** - Takes the output of the PID processes and tells each servo the angle it needs to steer to

Finally, we'll tune our PIDs independently and deploy the system.

18.2.1 Hardware Requirements

Figure 18.2: The PiMoroni Pan-Tilt HAT kit for a Raspberry Pi (http://pyimg.co/lh4on).

The goal of pan and tilt object tracking is for the camera to **stay centered upon an object.** Typically this tracking is accomplished with two servos. In our case, we have one servo for ***panning*** *left and right.* We have a separate servo for ***tilting*** *up and down*. Each of our servos and the fixture itself has a range of 180 degrees (some systems have a greater or lesser range than this).

You will need the following hardware to replicate this project:

- **Raspberry Pi** – I recommend the 3B+ or greater, but other models may work provided they have the same header pin layout.
- **PiCamera** – I recommend the PiCamera V2.
- **Pimoroni pan tilt HAT full kit** (http://pyimg.co/lh4on) The Pimoroni kit is a quality product and it hasn't let me down. Budget about 30 minutes for assembly. The SparkFun kit (http://pyimg.co/tsny0) would work as well, but it requires a soldering iron and additional assembly. Go for the Pimoroni kit if at all possible.
- **2.5A, 5V power supply** – If you supply less than 2.5A, your Pi might not have enough current, causing it to reset. Why? Because the servos draw necessary current away. Grab a **2.5A** power supply and dedicate it to this project hardware.

- **HDMI Screen** – Placing an HDMI screen next to your camera as you move around will allow you to visualize and debug, essential for manual tuning. Do not try X11 forwarding — it is simply too slow for video applications. VNC is possible if you don't have an HDMI screen.
- **Keyboard/mouse** - To accompany your HDMI screen.

18.2.2 A Brief Review on PIDs

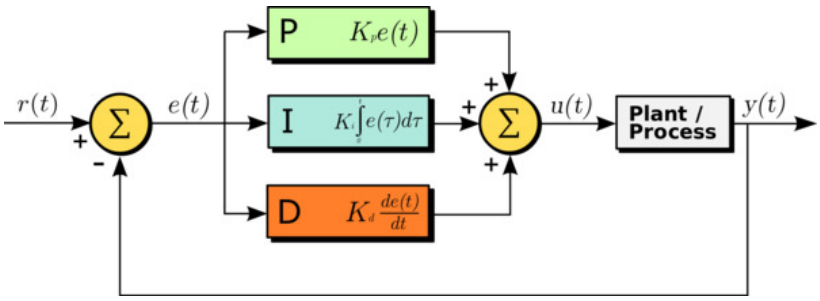

Figure 18.3: A Proportional Integral Derivative (PID) control loop will be used for each of our panning and tilting processes [67].

Be sure to refer to Chapter 17 for a thorough explanation of the Proportional Integral Derivative control loop (Figure 18.3). Let's review the essential concepts.

A PID has a target setpoint for the "process". The "process variable" is our feedback mechanism. Corrections are made to ensure that the process variable adjusts to meet the setpoint.

In this chapter, our "process variable" for panning is the x-coordinate of the center of the face. Our "process variable" for tilting is the y-coordinate of the center of the face. We will use the `PID` class and call the `.update` method inside of two independent control loops, to update the angles of both servos.

The result and goal are that our servos will move to keep the face centered in the frame. Recall the following when adjusting your PID gain constants:

- **P** – proportional, "present" (large corrections)
- **I** – integral, "in the past" (historical)
- **D** – derivative, "dampening" (anticipates the future)

18.2.3 Project Structure

Let's review the project structure:

```
|-- pyimagesearch
|   |-- __init__.py
|   |-- objcenter.py
|   |-- pid.py
|-- haarcascade_frontalface_default.xml
|-- home.py
|-- pan_tilt_tracking.py
```

In the remainder of this chapter, we'll be reviewing three Python files:

- `objcenter.py` : Calculates the center of a face bounding box using the Haar Cascade face detector. If you wish, you may detect a different type of object and place the logic in this file.
- `pan_tilt_tracking.py` : This is our pan/tilt object tracking driver script. It uses multi-processing with four independent processes (two of which are for panning and tilting, one is for finding an object, and one is for driving the servos with fresh angle values).

The `haarcascade_frontalface_default.xml` file is our pre-trained Haar Cascade face detector. Haar works great with the RPi in this context as it is much faster than HOG or Deep Learning. If your object detection method is not fast enough, then your camera tracking will lag behind the target.

18.2.4 Implementing the Face Detector and Object Center Tracker

The goal of our pan and tilt tracker will be to keep the camera centered on the object itself as shown in Figure 18.1. To accomplish this goal, we need to (1) detect the object itself, and (2) compute the center *(x, y)*-coordinates of the object.

Let's go ahead and implement our `ObjCenter` class inside `objcenter.py` which will accomplish both goals:

```
1 # import necessary packages
2 import cv2
3
4 class ObjCenter:
5     def __init__(self, haarPath):
6         # load OpenCV's Haar cascade face detector
7         self.detector = cv2.CascadeClassifier(haarPath)
```

We begin by importing OpenCV. Our `ObjCenter` class is defined on **Line 4**.

The constructor accepts a single argument — the path to the Haar Cascade face detector. Again, we're using the Haar method to find faces. Keep in mind that the RPi (even a 3B+ or

4) is a resource-constrained device. If you elect to use a slower (but more accurate) HOG or a CNN, the camera may not keep up with the moving object. You'll want to slow down the PID calculations so they aren't firing faster than you're actually detecting new face coordinates.

Remark. *You may also elect to use a Movidius NCS or Google Coral TPU USB Accelerator for deep learning face detection with the Raspberry Pi. Refer to the Hacker Bundle and Complete Bundle for examples on using these hardware accelerators.*

The `detector` itself is initialized on **Line 7**.

Let's define the `update` method which **finds the center** *(x, y)*-coordinate of a face:

```
9     def update(self, frame, frameCenter):
10        # convert the frame to grayscale
11        gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
12
13        # detect all faces in the input frame
14        rects = self.detector.detectMultiScale(gray, scaleFactor=1.05,
15            minNeighbors=9, minSize=(30, 30),
16            flags=cv2.CASCADE_SCALE_IMAGE)
17
18        # check to see if a face was found
19        if len(rects) > 0:
20            # extract the bounding box coordinates of the face and
21            # use the coordinates to determine the center of the
22            # face
23            (x, y, w, h) = rects[0]
24            faceX = int(x + (w / 2.0))
25            faceY = int(y + (h / 2.0))
26
27            # return the center (x, y)-coordinates of the face
28            return ((faceX, faceY), rects[0])
29
30        # otherwise no faces were found, so return the center of the
31        # frame
32        return (frameCenter, None)
```

Our project has *two* `update` *methods* so let's review the difference:

i. We previously reviewed the the `PID` class `update` method in Chapter 17. This method executes the PID algorithm to help calculate a new servo angle to keep the face/object in the center of the camera's field of view.

ii. Now we are reviewing the `ObjCenter` class `update` method. This method simply finds a face and returns its center current coordinates.

The `update` method (for finding the face) is defined on **Line 9** and accepts two parameters:

- `frame`: An image ideally containing one face
- `frameCenter`: The center coordinates of the frame

Line 12 converts the `frame` to grayscale (a preprocessing step for Haar object detection).

From there we **perform face detection** using the Haar Cascade `detectMultiScale` method. On **Lines 19-25** we check that faces have been detected and from there calculate the center *(x, y)*-coordinates of the face itself. **Lines 19-23** make an important assumption: we assume that only *one* face is in the `frame` at all times and that face can be accessed by the *0*-th index of `rects`.

Remark. *Without this assumption holding true, additional logic would be required to determine which face to track. See the section below on "Improvements for pan/tilt face tracking with the Raspberry Pi", where I describe how to handle multiple face detections with Haar.*

The center of the face, as well as the bounding box coordinates, are returned on **Line 28**. We'll use the bounding box coordinates to draw a box around the face for display purposes.

Otherwise, when no faces are found, we simply return the center of the frame on **Line 32** (ensuring that the servos stop and do not make any corrections until a face is found again).

18.2.5 The Pan and Tilt Driver Script

Now that we have a system in place to find the center of a face as well as our PID class, let's work on the driver script to tie it all together. Go ahead and create a new file called `pan_tilt_tracking.py` and insert these lines:

```
1  # import necessary packages
2  from pyimagesearch.objcenter import ObjCenter
3  from pyimagesearch.pid import PID
4  from multiprocessing import Manager
5  from multiprocessing import Process
6  from imutils.video import VideoStream
7  import pantilthat as pth
8  import argparse
9  import signal
10 import time
11 import sys
12 import cv2
13
14 # define the range for the motors
15 servoRange = (-90, 90)
```

On **Line 2-12** we import necessary libraries:

- `Process` and `Manager` will help us with `multiprocessing` and shared variables.
- `VideoStream` will allow us to grab frames from our camera
- `ObjCenter` will help us locate the object in the frame, while `PID` will help us keep the object in the center of the frame by calculating our servo angles
- `pantilthat` is the library used to interface with the RPi's Pimoroni pan tilt HAT

Our servos on the pan tilt HAT have a range of 180 degrees (-90 to 90) as is defined on **Line 15**. These values should reflect the limitations of your servos or any soft limits you'd like to put in place (i.e. if you don't want your camera to aim at the floor, wall, or ceiling).

Let's define a *"ctrl + c"* `signal_handler`:

```
17 # function to handle keyboard interrupt
18 def signal_handler(sig, frame):
19     # disable the servos and gracefully exit
20     print("[INFO] You pressed `ctrl + c`! Exiting...")
21     pth.servo_enable(1, False)
22     pth.servo_enable(2, False)
23     sys.exit()
```

This multiprocessing script can be tricky to exit from. There are a number of ways to accomplish it, but I decided to go with a `signal_handler` approach (first introduced in Chapter 7).

Inside the signal handler, **Line 20** prints a status message, **Lines 21 and 22** disable our servos, and **Line 23** exits from our program.

You might look at this script as a whole and think *"If I have four processes, and `signal_handler` is running in each of them, then this will occur four times."*

You are absolutely right, but this is a compact and understandable way to go about killing off our processes, short of pressing *"ctrl + c"* as many times as you can in a sub-second period to try to get all processes to die off. Imagine if you had 10 processes and were trying to kill them with the *"ctrl + c"* approach. There are certainly better approaches presented online, and I encourage you to investigate them.

Now that we know how our processes will exit, let's define our first process:

```
25 def obj_center(args, objX, objY, centerX, centerY):
26     # set the signal trap to handle keyboard interrupt and then
27     # initialize the object center finder
28     signal.signal(signal.SIGINT, signal_handler)
29     obj = ObjCenter(args["cascade"])
```

```
30
31      # start the video stream and wait for the camera to warm up
32      vs = VideoStream(usePiCamera=True).start()
33      time.sleep(2.0)
34
35      # loop indefinitely
36      while True:
37          # grab the frame from the threaded video stream and flip it
38          # vertically (since our camera was upside down)
39          frame = vs.read()
40          frame = cv2.flip(frame, 0)
41
42          # calculate the center of the frame as this is (ideally) where
43          # we will we wish to keep the object
44          (H, W) = frame.shape[:2]
45          centerX.value = W // 2
46          centerY.value = H // 2
47
48          # find the object's location
49          objectLoc = obj.update(frame, (centerX.value, centerY.value))
50          ((objX.value, objY.value), rect) = objectLoc
51
52          # extract the bounding box and draw it on the frame
53          if rect is not None:
54              (x, y, w, h) = rect
55              cv2.rectangle(frame, (x, y), (x + w, y + h),
56                  (0, 255, 0), 2)
57
58          # display the frame to the screen
59          cv2.imshow("Pan-Tilt Face Tracking", frame)
60          cv2.waitKey(1)
```

Our `obj_center` thread begins on **Line 29** and accepts five parameters:

- `args`: Our command line arguments dictionary (created in our main thread).
- `objX` and `objY`: The *(x, y)*-coordinates of the object. We'll continuously calculate these values.
- `centerX` and `centerY`: The center of the frame.

On **Line 28** we start our `signal_handler`.

Our `ObjCenter` is instantiated as `obj` on **Line 29**. Our cascade path is passed to the constructor.

Then, on **Lines 32 and 33**, we start our `VideoStream` for our `PiCamera`, allowing it to warm up for two seconds.

From here, our process enters an infinite loop on **Line 36**. The only way to escape out of the loop is if the user types *"ctrl + c"* (you'll notice the lack of no `break` statement).

Our `frame` is grabbed and flipped on **Lines 39 and 40**. We must `flip` the `frame` because the PiCamera is physically upside down in the pan tilt HAT fixture by design.

Lines 44-46 set our frame width and height as well as calculate the center point of the frame. You'll notice that we are using `.value` to access our center point variables — this is required with the `Manager` method of sharing data between processes.

To calculate where our object is, we'll simply call the `update` method on `obj` while passing the video `frame`. The reason we also pass the center coordinates is because we'll just have the `ObjCenter` class return the frame center if it doesn't see a Haar face. Effectively, this makes the PID error `0` and thus, the servos stop moving and remain in their current positions until a face is found.

Remark. *I choose to return the frame center if the face could not be detected. Alternatively, you may wish to return the coordinates of the last location a face was detected. That is an implementation choice that I leave up to you.*

The result of the `update` (**Line 49**) is parsed on **Line 50** where our object coordinates and the bounding box are assigned.

The last steps are to (1) draw a rectangle around our face and (2) display the video frame (**Lines 53-60**).

Let's define our next process, `pid_process`:

```
62 def pid_process(output, p, i, d, objCoord, centerCoord):
63     # set the signal trap to handle keyboard interrupt, then create a
64     # PID and initialize it
65     signal.signal(signal.SIGINT, signal_handler)
66     p = PID(p.value, i.value, d.value)
67     p.initialize()
68
69     # loop indefinitely
70     while True:
71         # calculate the error and update the vlaue
72         error = centerCoord.value - objCoord.value
73         output.value = p.update(error)
```

Our `pid_process` is quite simple as the heavy lifting is taken care of by the PID class. Two of these processes will be running at any given time (panning and tilting). If you have a complex robot, you might have many more PID processes running. The method accepts six parameters:

- `output`: The servo angle that is calculated by our PID controller. This will be a pan or tilt angle.

- `p`, `i`, and `d`: Our three PID constants.
- `objCoord`: This value is passed to the process so that the process has access to keep track of where the object is. For panning, it is an *x*-coordinate. For tilting, it is a *y*-coordinate.
- `centerCoord`: Used to calculate our `error`, this value is just the center of the frame (either *x* or *y* depending on whether we are panning or tilting).

Be sure to trace each of the parameters back to where the process is started in the main thread of this program.

On **Line 65**, we start our `signal_handler`.

Then we instantiate our `PID` on **Line 65**, passing each of the P, I, and D values. Subsequently, the `PID` object is initialized (**Line 67**).

Now comes the fun part in just two lines of code:

i. Calculate the `error` on **Line 72**. For example, this could be the frame's *y*-center minus the object's *y*-location for tilting.

ii. Call `update` (**Line 73**), passing the new error (and a sleep time if necessary). The returned value is the `output.value` (an angle in degrees).

We have another thread that "watches" each `output.value` to drive the servos. Speaking of driving our servos, let's implement a servo range checker and our servo driver now:

```
75 def in_range(val, start, end):
76     # determine the input vale is in the supplied range
77     return (val >= start and val <= end)
78
79 def set_servos(pan, tlt):
80     # set the signal trap to handle keyboard interrupt
81     signal.signal(signal.SIGINT, signal_handler)
82
83     # loop indefinitely
84     while True:
85         # the pan and tilt angles are reversed so multiply by -1
86         panAngle = -1 * pan.value
87         tltAngle = -1 * tlt.value
88
89         # if the pan angle is within the range, pan
90         if in_range(panAngle, servoRange[0], servoRange[1]):
91             pth.pan(panAngle)
92
93         # if the tilt angle is within the range, tilt
```

```
94          if in_range(tltAngle, servoRange[0], servoRange[1]):
95              pth.tilt(tltAngle)
```

Lines 75-77 define an `in_range` method to determine if a value is within a particular range.

From there, we'll drive our servos to specific pan and tilt angles in the `set_servos` method. This method will be running in another process. It accepts `pan` and `tlt` values, and will watch the values for updates. The values themselves are constantly being adjusted via our `pid_process`.

We initialize our `signal_handler` for this process on **Line 81**.

From there, we'll start our infinite loop until a signal is caught:

- Our `panAngle` and `tltAngle` values are made negative to accommodate the orientation of the servos and camera (**Lines 86 and 87**)
- Then we check each value ensuring it is in the range as well as drive the servos to the new angle (**Lines 90-95**)

That was easy.

Now let's parse command line arguments:

```
97  # check to see if this is the main body of execution
98  if __name__ == "__main__":
99      # construct the argument parser and parse the arguments
100     ap = argparse.ArgumentParser()
101     ap.add_argument("-c", "--cascade", type=str, required=True,
102         help="path to input Haar cascade for face detection")
103     args = vars(ap.parse_args())
```

The main body of execution begins on **Line 98**.

We parse our command line arguments on **Lines 100-103**. We only have one — the path to the Haar Cascade on disk.

Now let's work with process-safe variables and start each independent process:

```
105     # start a manager for managing process-safe variables
106     with Manager() as manager:
107         # enable the servos
108         pth.servo_enable(1, True)
109         pth.servo_enable(2, True)
110
```

```
111         # set integer values for the object center (x, y)-coordinates
112         centerX = manager.Value("i", 0)
113         centerY = manager.Value("i", 0)
114
115         # set integer values for the object's (x, y)-coordinates
116         objX = manager.Value("i", 0)
117         objY = manager.Value("i", 0)
118
119         # pan and tilt values will be managed by independent PIDs
120         pan = manager.Value("i", 0)
121         tlt = manager.Value("i", 0)
```

Inside the `Manager` block, our process safe variables are established. We have quite a few of them.

First, we enable the servos on **Lines 108 and 109**. Without these lines, the hardware won't work.

Let's look at our first handful of process safe variables:

- The frame center coordinates are integers (denoted by `"i"`) and initialized to `0` (**Lines 112 and 111**).
- The object center coordinates, also integers and initialized to `0` (**Lines 116 and 117**).
- Our `pan` and `tlt` angles (**Lines 120 and 121**) are integers that I've set to start in the center, pointing towards a face (angles of `0` degrees).

Now we'll set the P, I, and D constants:

```
123         # set PID values for panning
124         panP = manager.Value("f", 0.09)
125         panI = manager.Value("f", 0.08)
126         panD = manager.Value("f", 0.002)
127
128         # set PID values for tilting
129         tiltP = manager.Value("f", 0.11)
130         tiltI = manager.Value("f", 0.10)
131         tiltD = manager.Value("f", 0.002)
```

Our panning and tilting PID constants (process safe) are set on **Lines 124-131**. These are floats. Be sure to review the PID tuning section (Section 18.2.6) next to learn how we found suitable values. **To get the most value out of this project, I would recommend setting each to zero and following the tuning method/process** (not to be confused with a computer science method/process).

With all of our process safe variables ready to go, let's launch our processes:

```
133         # we have 4 independent processes
134         # 1. objectCenter  - finds/localizes the object
135         # 2. panning       - PID control loop determines panning angle
136         # 3. tilting       - PID control loop determines tilting angle
137         # 4. setServos     - drives the servos to proper angles based
138         #                    on PID feedback to keep object in center
139         processObjectCenter = Process(target=obj_center,
140             args=(args, objX, objY, centerX, centerY))
141         processPanning = Process(target=pid_process,
142             args=(pan, panP, panI, panD, objX, centerX))
143         processTilting = Process(target=pid_process,
144             args=(tlt, tiltP, tiltI, tiltD, objY, centerY))
145         processSetServos = Process(target=set_servos, args=(pan, tlt))
146
147         # start all 4 processes
148         processObjectCenter.start()
149         processPanning.start()
150         processTilting.start()
151         processSetServos.start()
152
153         # join all 4 processes
154         processObjectCenter.join()
155         processPanning.join()
156         processTilting.join()
157         processSetServos.join()
158
159         # disable the servos
160         pth.servo_enable(1, False)
161         pth.servo_enable(2, False)
```

Each process is defined on **Lines 139-145**, passing required process safe values. We have four processes:

i. A process which **finds the object in the frame**. In our case, it is a face.

ii. A process which **calculates panning (left and right) angles** with a **PID**.

iii. A process which **calculates tilting (up and down) angles** with a **PID**.

iv. A process which **drives the servos**.

Each of the processes is started and then joined (**Lines 148-157**).

Servos are disabled when all processes exit (**Lines 160 and 161**). This also occurs in the `signal_handler` for when our program receives an interrupt signal.

18.2.6 Manual Tuning

That was a lot of work, but we're not done yet. Now that we understand the code, we need to perform manual tuning of our two independent PIDs (one for panning and one for tilting).

Tuning a PID ensures that our servos will track the object (in our case, a face) smoothly.

Be sure to refer to the *"How to tune a PID"* section from the previous chapter (Section 17.3.3). Additionally, the manual tuning section of Wikipedia's PID article is a great resource [67].

You should follow this process:

i. Set `kI` and `kD` to zero.

ii. Increase `kP` from zero until the output oscillates (i.e. the servo goes back and forth or up and down). Then set the value to half.

iii. Increase `kI` until offsets are corrected quickly, knowing that a value that is too high will cause instability.

iv. Increase `kD` until the output settles on the desired output reference quickly after a load disturbance (i.e. if you move your face somewhere really fast). Too much `kD` will cause excessive response and make your output overshoot where it needs to be.

I cannot stress this enough: *Make small changes while tuning.*

We will be tuning our PIDs independently, first by **tuning the tilting process.**

Go ahead and ***comment out*** **the panning process** in the driver script:

```
147         # start all 4 processes
148         processObjectCenter.start()
149         #processPanning.start()
150         processTilting.start()
151         processSetServos.start()
152
153         # join all 4 processes
154         processObjectCenter.join()
155         #processPanning.join()
156         processTilting.join()
157         processSetServos.join()
```

From there, open up a terminal and execute the following command:

```
$ python pan_tilt_tracking.py --cascade \
    haarcascade_frontalface_default.xml
```

You will need to follow the manual tuning guide above to tune the tilting process. While doing so, you'll need to:

i. Start the program and move your face up and down, causing the camera to tilt. I recommend doing squats at your knees while looking directly at the camera.

ii. Stop the program and adjust values, per the tuning guide.

iii. Repeat **until you're satisfied with the result** (and thus, the values). It should be tilting well with small displacements, as well as large changes, in where your face is. Be sure to test both.

At this point, let's **switch to the other PID.** The values will be similar, but it is necessary to tune them as well. Go ahead and ***comment out* the tilting process** (which is fully tuned) and **uncomment the panning process:**

```
147         # start all 4 processes
148         processObjectCenter.start()
149         processPanning.start()
150         #processTilting.start()
151         processSetServos.start()
152
153         # join all 4 processes
154         processObjectCenter.join()
155         processPanning.join()
156         #processTilting.join()
157         processSetServos.join()
```

And once again, execute the following command:

```
$ python pan_tilt_tracking.py \
    --cascade haarcascade_frontalface_default.xml
```

Now follow the steps above again to **tune the panning process**, only this time moving from side to side rather than up and down.

18.2.7 Run Panning and Tilting Processes at the Same Time

With our freshly tuned PID constants, let's put our pan and tilt camera to the test.

Assuming you followed the section above, ensure that both processes (panning and tilting) are uncommented and ready to go.

From there, open up a terminal and execute the following command:

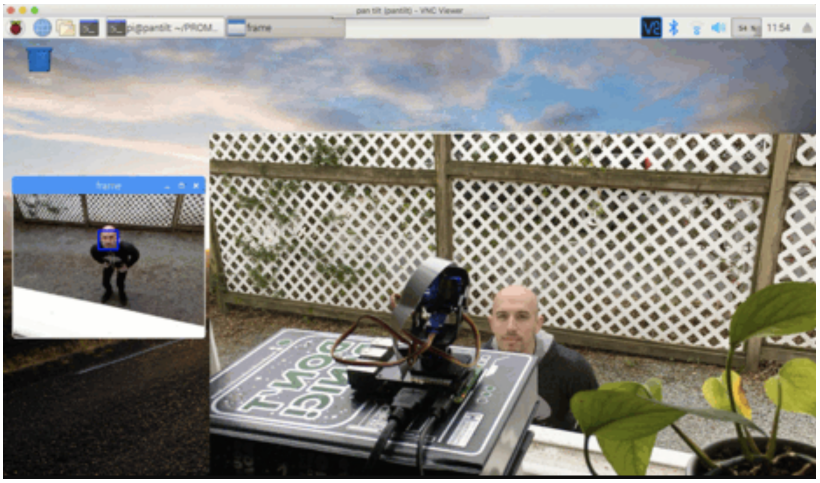

Figure 18.4: Pan/tilt face tracking demonstration. Full GIF demo available here: http://pyimg.co/pb9f2

```
$ python pan_tilt_tracking.py \
    --cascade haarcascade_frontalface_default.xml
```

Once the script is up and running you can walk in front of your camera. If all goes well, you should see your face being *detected* and *tracked* as shown in Figure 18.4. Click this link to see a demo in your browser: http://pyimg.co/pb9f2.

18.3 Improvements for Pan/Tilt Tracking with the Raspberry Pi

There are times when the camera will encounter a false positive face, causing the control loop to go haywire. Don't be fooled! Your PID is working just fine, but your computer vision environment is impacting the system with false information.

We chose Haar because it is fast; however, Haar *can* lead to false positives:

- Haar isn't as accurate as HOG. HOG is great, but is resource hungry compared to Haar.
- Haar is far from accurate compared to a Deep Learning face detection method. The DL method is too slow to run on the Pi and real-time. If you tried to use the DL face detector, panning and tilting would be pretty jerky.

My recommendation is that you set up your pan/tilt camera in a new environment and see if that improves the results. For example, when we were testing the face tracking, we found that it didn't work well in a kitchen due to reflections off the floor, refrigerator, etc.

Then when we aimed the camera out the window and I stood outside, and the tracking improved drastically, because `ObjCenter` was providing legitimate values for the face. Thus our PID could do its job.

What if there are two faces in the frame? Or what if I'm the only face in the frame, but consistently there is a false positive?

This is a great question. In general, you'd want to track only one face, so there are a number of options:

- Use the **confidence value and take the face with the highest confidence.** This is **not possible using the default Haar detector code** as it doesn't report confidence values. Instead, let's explore other options.
- Try to get the `rejectLevels` and `rejectWeights` from the Haar cascade detections. I've never tried this, but the following links may help:
  - OpenCV Answers thread: http://pyimg.co/iej12
  - StackOverflow thread: http://pyimg.co/x2f6e
- **Grab the largest bounding box** — easy and simple.
- **Select the face closest to the center of the frame.** Since the camera tries to keep the face closest to the center, we could compute the Euclidean distance between all centroid bounding boxes and the center *(x, y)*-coordinates of the frame. The bounding box closest to the centroid would be selected.

18.4 Summary

In this chapter, we learned how to build a pan/tilt tracking system with our Raspberry Pi.

The system is capable of tracking any object provided that the Pi has enough computational horsepower to find the object. We took advantage of the fast Haar Cascade algorithm to find and track our face. We then deployed two PIDs using our `PID` class and multiprocessing. Finally we learned out to tune PID constants — a critical step. The result is a relatively smooth tracking algorithm!

Finally, be sure to refer to the PyImageSearch blog post from April 2019 on the topic as well (http://pyimg.co/aiqr5) [71]. The *"Comments"* section in particular provides a good discussion on pan/tilt tips, tricks, and suggestions from other PyImageSearch readers.

This case study on *“Creating a People/Footfall Counter”* chapter is from the *Hobbyist Bundle*…

Chapter 19

Creating a People/Footfall Counter

A highly requested topic by readers of the PyImageSearch blog has been to create a people counter application. In August 2018, I wrote such a tutorial, entitled *OpenCV People Counter* (http://pyimg.co/vgak6) [72].

The tutorial was based upon a (1) a pre-trained MobileNet SSD object detector, (2) dlib's correlation tracker (http://pyimg.co/yqlsy) [73], and (3) my implementation of centroid tracking (http://pyimg.co/nssn6) [74].

Soon after the post published, readers asked a followup question:

> *"I want to track people entering/exiting my store, but I want to keep costs down and mount an RPi near the doorway. Can the* OpenCV People Counter *run on a Raspberry Pi?"*

More courageous readers even tried out the code on a Raspberry Pi 3B+ with no modifications. They were disheartened by the fact that the RPi 3B+ achieved a minuscule 4.89 FPS making the methodology unusable — the people moved in "slow motion" using a video file as input and had we deployed this in the field, people would not have been counted properly.

That's not to say it is impossible to count people using an RPi though!

Working on resource-constrained hardware requires that you bring the right weapons to the fight. In the case of using the Raspberry Pi CPU for people counting, your nunchucks are background subtraction and your dagger is Haar Cascade. In this chapter, we will learn to make a handful of modifications to the original people tracking system I presented.

19.1 Chapter Learning Objectives

In this chapter we will learn:

i. Saving CPU cycles with the efficient background subtraction algorithm

ii. Object detection tradeoffs and how Haar Cascades are a great alternative

iii. Object tracking tradeoffs for the Raspberry Pi

iv. How to put these concepts together to build a reliable people counting application that will run on your Raspberry Pi

19.2 Background Subtraction + Haar Based People Counter

In this section, we'll briefly review our project structure. From there, we'll develop our `TrackableObject`, `CentroidTracker`, and `DirectionCounter` classes. We'll then implement the classes into our `people_counter.py` driver script which is optimized and tweaked for the Raspberry Pi. Finally, we'll learn how to execute the program; we'll report statistics for both the RPi 3B+ and RPi 4.

19.2.1 Project Structure

```
|-- videos
|   |-- horizontal_01.avi
|   |-- vertical_01.mp4
|   |-- vertical_02.mp4
|-- output
|   |-- output_01.avi
|-- pyimagesearch
|   |-- __init__.py
|   |-- centroidtracker.py
|   |-- directioncounter.py
|   |-- trackableobject.py
|-- people_counter.py
```

Input videos for testing are included in the `videos/` directory. The input videos are provided by David McDuffee.

Our `output/` folder will be where we'll store processed videos. One example output video is included.

The `pyimagesearch` module contains three classes which we will review: (1) `TrackableObject`, (2) `CentroidTracker`, and (3) `DirectionCounter`. Each trackable object is assigned an ID number. The centroid tracker associates objects and updates the trackable objects. Each trackable object has a list of centroids — its current location centroid, and all previous locations in the frame. The direction counter analyzes the current and historical lo-

cations to determine which direction an object is moving in. It also counts an object if it has passed certain horizontal or vertical line in the frame.

We'll also walk through the driver script in detail, `people_counter.py`. This script takes advantage of all the aforementioned classes in order to count people on a resource-constrained device.

19.2.2 Centroid Tracking

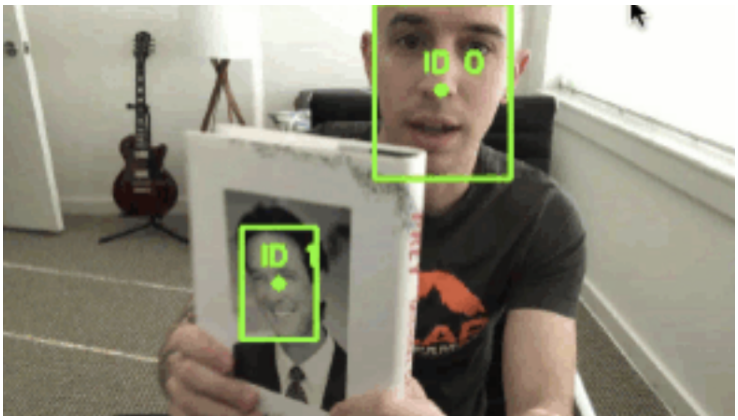

Figure 19.1: An example of centroid tracking in action. Notice how each detected face has a unique integer IDs associated with it. A full GIF of the demo can be found here: http://pyimg.co/jhuwj

I first presented a simple object tracker in a blog post in July 2018 (http://pyimg.co/nssn6) [74]. Object tracking, is arguably one of the most requested topics here on PyImageSearch. **Object tracking is the process of:**

i. Taking an initial set of object detections (such as an input set of bounding box coordinates)

ii. Creating a unique ID for each of the initial detections

iii. And then tracking each of the objects as they move around frames in a video, maintaining the assignment of unique IDs

Furthermore, object tracking allows us to **apply a unique ID to each tracked object**, making it possible for us to count unique objects in a video. Object tracking is paramount to building a **person counter.**

An ideal object tracking algorithm will:

- Only require the object detection phase once (i.e., when the object is initially detected)

- Be extremely fast — *much* faster than running the actual object detector itself
- Be able to handle when the tracked object "disappears" or moves outside the boundaries of the video frame
- Be robust to occlusion
- Be able to pick up objects it has "lost" in between frames

This is a tall order for any computer vision or image processing algorithm, and there are a variety of tricks we can play to help improve our object trackers on resource-constrained devices. But before we can build such a robust method, we first need to study the fundamentals of object tracking.

19.2.2.1 Fundamentals of Object Tracking

A simple object tracking algorithm relies on keeping track of the centroids of objects.

Typically an *object tracker* works hand-in-hand with a less-efficient *object detector*. The *object detector* is responsible for localizing an object. The *object tracker* is responsible for keeping track of which object is which by assigning and maintaining identification numbers (IDs).

This object tracking algorithm we're implementing is called *centroid tracking* as it relies on the Euclidean distance between (1) *existing* object centroids (i.e., objects the centroid tracker has already seen before), and (2) new object centroids between subsequent frames in a video.

We'll review the centroid algorithm in more depth going forward.

19.2.2.2 The Centroid Tracking Algorithm

The centroid tracking algorithm is a multi-step process. The five steps include:

i. **Step #1:** Accept bounding box coordinates and compute centroids
ii. **Step #2:** Compute Euclidean distance between new bounding boxes and existing objects
iii. **Step #3:** Update *(x, y)*-coordinates of existing objects
iv. **Step #4:** Register new objects
v. **Step #5:** Deregister old/lost objects that have moved out of frame

Let's review each of these steps in more detail.

19.2.2.2,1 Step #1: Accept Bounding Box Coordinates and Compute Centroids

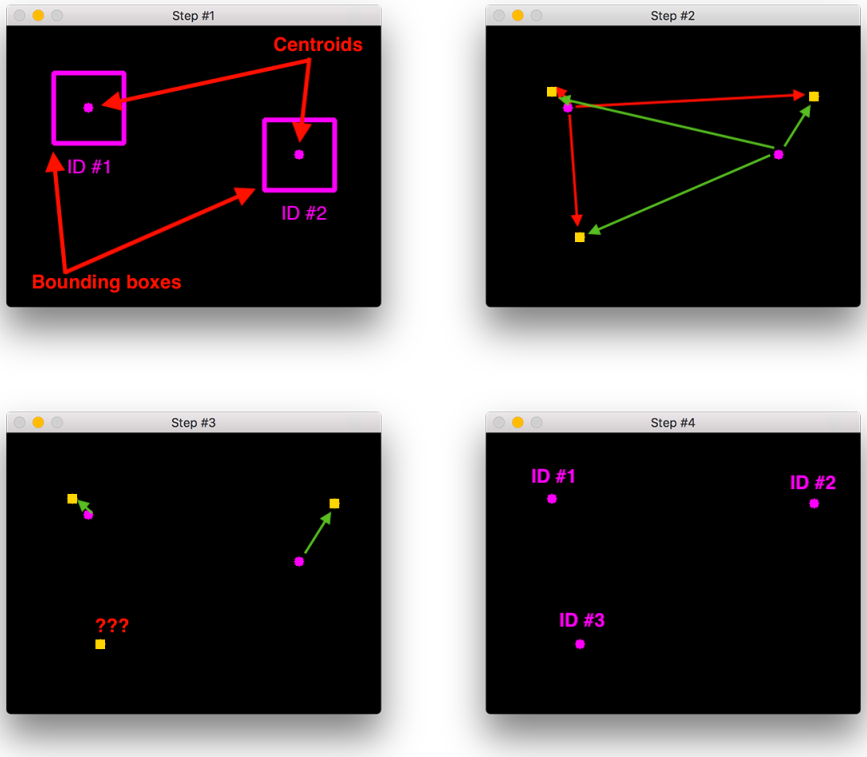

Figure 19.2: **Top-left:** To build a simple object tracking algorithm using centroid tracking, the first step is to accept bounding box coordinates from an object detector and use them to compute centroids. **Top-right:** In the next input frame, three objects are now present. We need to compute the Euclidean distances between each pair of original centroids (*circle*) and new centroids (*square*). **Bottom-left:** Our simple centroid object tracking method has associated objects with minimized object distances. What do we do about the object in the bottom left though? **Bottom-right:** We have a new object that wasn't matched with an existing object, so it is registered as object ID #3.

The centroid tracking algorithm assumes that we are passing in a set of bounding box *(x, y)*-coordinates for each detected object in ***every single frame***. These bounding boxes can be produced by any type of object detector you would like (color thresholding + contour extraction, Haar cascades, HOG + Linear SVM, SSDs, Faster R-CNNs, etc.), provided that they are computed for every frame in the video.

Once we have the bounding box coordinates we must compute the "centroid", or more simply, the center *(x, y)*-coordinates of the bounding box. Figure 19.2 *(top-left)* demonstrates accepting a set of bounding box coordinates and computing the centroid.

Since these are the first initial set of bounding boxes presented to our algorithm we will assign them unique IDs.

19.2.2.2,2 Step #2: Compute Euclidean Distance Between New Bounding Boxes and Existing Objects

For every subsequent frame in our video stream, we apply **Step #1** of computing object centroids; however, instead of assigning a new unique ID to each detected object (which would defeat the purpose of object tracking), we first need to determine if we can *associate* the *new* object centroids (circles) with the *old* object centroids (squares). To accomplish this process, we compute the Euclidean distance (highlighted with green arrows) between each pair of existing object centroids and input object centroids.

From Figure 19.2 *(top-right)* you can see that we have this time detected three objects in our image. The two pairs that are close together are two existing objects. We then compute the Euclidean distances between each pair of original centroids (yellow) and new centroids (purple).

But how do we use the Euclidean distances between these points to actually match them and associate them?

The answer is in **Step #3**.

19.2.2.2,3 Step #3: Update *(x, y)*-coordinates of Existing Objects

The primary assumption of the centroid tracking algorithm is that a given object will potentially move in between subsequent frames, but the *distance* between the centroids for frames F_t and F_{t+1} will be *smaller* than all other distances between objects.

Therefore, if we choose to associate centroids with minimum distances between subsequent frames we can build our object tracker.

In Figure 19.2 *(bottom-right)* you can see how our centroid tracker algorithm chooses to associate centroids that minimize their respective Euclidean distances.

But what about the lonely point in the bottom-left? It didn't get associated with anything — what do we do with it?

19.2.2.2,4 Step #4: Register New Objects

In the event that there are more input detections than existing objects being tracked, we need to register the new object. "Registering" simply means that we are adding the new object to our list of tracked objects by (1) assigning it a new object ID, and (2) storing the centroid of the bounding box coordinates for that object

We can then go back to **Step #2** and repeat the pipeline of steps for every frame in our video stream.

Figure 19.2 *(bottom-right)* demonstrates the process of using the minimum Euclidean distances to associate existing object IDs and then registering a new object.

19.2.2.2,5 Step #5: Deregister Old Objects

Any reasonable object tracking algorithm needs to be able to handle when an object has been lost, disappeared, or left the field of view. Exactly how you handle these situations is really dependent on where your object tracker is meant to be deployed, but for this implementation, we will deregister old objects when they cannot be matched to any existing objects for a total of N subsequent frames.

19.2.2.3 Centroid Tracking Implementation

Before we can apply object tracking to our input video streams, we first need to implement the centroid tracking algorithm. While you're digesting this centroid tracker script, just keep in mind **Steps #1 through Step #5 detailed above** and review the steps as necessary. As you'll see, the translation of steps to code requires quite a bit of thought, and while we perform all steps, they aren't linear due to the nature of our various data structures and code constructs.

I would suggest re-reading the steps above, re-reading the code explanation for the centroid tracker, and finally reviewing it all once more. This will bring everything full circle, and allow you to wrap your head around the algorithm.

Once you're sure you understand the steps in the centroid tracking algorithm, open up the `centroidtracker.py` inside the `pyimagesearch` module, and let's review the code:

```
1  # import the necessary packages
2  from scipy.spatial import distance as dist
3  from collections import OrderedDict
4  import numpy as np
5
6  class CentroidTracker:
```

```
7       def __init__(self, maxDisappeared=50, maxDistance=50):
8           # initialize the next unique object ID along with two ordered
9           # dictionaries used to keep track of mapping a given object
10          # ID to its centroid and number of consecutive frames it has
11          # been marked as "disappeared", respectively
12          self.nextObjectID = 0
13          self.objects = OrderedDict()
14          self.disappeared = OrderedDict()
15
16          # store the number of maximum consecutive frames a given
17          # object is allowed to be marked as "disappeared" until we
18          # need to deregister the object from tracking
19          self.maxDisappeared = maxDisappeared
20
21          # store the maximum distance between centroids to associate
22          # an object -- if the distance is larger than this maximum
23          # distance we'll start to mark the object as "disappeared"
24          self.maxDistance = maxDistance
```

On **Lines 2-4** we import our required packages and modules — `distance`, `OrderedDict`, and `numpy`.

Our `CentroidTracker` class is defined on **Line 6**. The constructor accepts a single parameter, the maximum number of consecutive frames a given object has to be lost/disappeared for until we remove it from our tracker (**Line 7**). Our constructor builds five class variables:

- `nextObjectID`: A counter used to assign unique IDs to each object (**Line 12**). In the case that an object leaves the frame and does not come back for `maxDisappeared` frames, a new (next) object ID would be assigned.
- `objects`: A dictionary that utilizes the object ID as the key and the centroid *(x, y)*-coordinates as the value (**Line 13**).
- `disappeared`: Maintains number of consecutive frames (value) a particular object ID (key) has been marked as "lost" (**Line 14**).
- `maxDisappeared`: The number of consecutive frames an object is allowed to be marked as "lost/disappeared" until we deregister the object.
- `maxDistance`: The maximum distance between centroids to associate an object. If the distance exceeds this value, then the object will be marked as disappeared.

Let's define the `register` method which is responsible for adding new objects to our tracker:

```
26      def register(self, centroid):
27          # when registering an object we use the next available object
```

```
28          # ID to store the centroid
29          self.objects[self.nextObjectID] = centroid
30          self.disappeared[self.nextObjectID] = 0
31          self.nextObjectID += 1
32
33      def deregister(self, objectID):
34          # to deregister an object ID we delete the object ID from
35          # both of our respective dictionaries
36          del self.objects[objectID]
37          del self.disappeared[objectID]
```

The `register` method is defined on **Line 26**. It accepts a centroid and then adds it to the `objects` dictionary using the next available object ID.

The number of times an object has disappeared is initialized to `0` in the `disappeared` dictionary (**Line 30**).

Finally, we increment the `nextObjectID` so that if a new object comes into view, it will be associated with a unique ID (**Line 31**).

Similar to our registration method, we also need a `deregister` method. Just like we can add new objects to our tracker, we also need the ability to remove old ones that have been lost or disappeared from our input frames themselves. The `deregister` method is defined on **Line 33**. It simply deletes the `objectID` in both the `objects` and `disappeared` dictionaries, respectively (**Lines 36 and 37**).

The heart of our centroid tracker implementation lives inside the `update` method:

```
39      def update(self, rects):
40          # check to see if the list of input bounding box rectangles
41          # is empty
42          if len(rects) == 0:
43              # loop over any existing tracked objects and mark them
44              # as disappeared
45              for objectID in list(self.disappeared.keys()):
46                  self.disappeared[objectID] += 1
47
48                  # if we have reached a maximum number of consecutive
49                  # frames where a given object has been marked as
50                  # missing, deregister it
51                  if self.disappeared[objectID] > self.maxDisappeared:
52                      self.deregister(objectID)
53
54              # return early as there are no centroids or tracking info
55              # to update
56              return self.objects
57
58          # initialize an array of input centroids for the current frame
59          inputCentroids = np.zeros((len(rects), 2), dtype="int")
```

```
60
61          # loop over the bounding box rectangles
62          for (i, (startX, startY, endX, endY)) in enumerate(rects):
63              # use the bounding box coordinates to derive the centroid
64              cX = int((startX + endX) / 2.0)
65              cY = int((startY + endY) / 2.0)
66              inputCentroids[i] = (cX, cY)
```

The `update` method, defined on **Line 39**, accepts a list of bounding box rectangles, presumably from an object detector (Haar cascade, HOG + Linear SVM, SSD, Faster R-CNN, etc.). The format of the `rects` parameter is assumed to be a tuple with this structure: `(startX, startY, endX, endY)`.

If there are no detections, we'll loop over all object IDs and increment their `disappeared` count (**Lines 42-46**). We'll also check if we have reached the maximum number of consecutive frames a given object has been marked as missing. If that is the case we need to remove it from our tracking systems (**Lines 51 and 52**). Since there is no tracking info to update, we go ahead and `return` early on **Line 51**. Otherwise, we have quite a bit of work to do to complete our `update` method implementation.

On **Line 59** we'll initialize a NumPy array to store the centroids for each `rect`.

Then, we loop over bounding box rectangles (**Line 62**) and compute the centroid and store it in the `inputCentroids` list (**Lines 64-66**).

If there are currently no objects we are tracking, we'll `register` each of the new objects:

```
68          # if we are currently not tracking any objects take the input
69          # centroids and register each of them
70          if len(self.objects) == 0:
71              for i in range(0, len(inputCentroids)):
72                  self.register(inputCentroids[i])
```

Otherwise, we need to update any existing object *(x, y)*-coordinates based on the centroid location that minimizes the Euclidean distance between them:

```
74          # otherwise, are are currently tracking objects so we need to
75          # try to match the input centroids to existing object
76          # centroids
77          else:
78              # grab the set of object IDs and corresponding centroids
79              objectIDs = list(self.objects.keys())
80              objectCentroids = list(self.objects.values())
81
82              # compute the distance between each pair of object
83              # centroids and input centroids, respectively -- our
```

```
84              # goal will be to match an input centroid to an existing
85              # object centroid
86              D = dist.cdist(np.array(objectCentroids), inputCentroids)
87
88              # in order to perform this matching we must (1) find the
89              # smallest value in each row and then (2) sort the row
90              # indexes based on their minimum values so that the row
91              # with the smallest value as at the *front* of the index
92              # list
93              rows = D.min(axis=1).argsort()
94
95              # next, we perform a similar process on the columns by
96              # finding the smallest value in each column and then
97              # sorting using the previously computed row index list
98              cols = D.argmin(axis=1)[rows]
```

The updates to existing tracked objects take place beginning at the `else` on **Line 77**. The goal is to track the objects and to maintain correct object IDs — this process is accomplished by computing the Euclidean distances between all pairs of `objectCentroids` and `inputCentroids`, followed by associating object IDs that minimize the Euclidean distance.

Inside of the `else` block, beginning on **Line 77**, we will first grab `objectIDs` and `objectCentroid` values (**Lines 79 and 80**). We then compute the distance between each pair of existing object centroids and new input centroids (**Line 86**). The output NumPy array shape of our distance map `D` will be `(# of object centroids, # of input centroids)`.

To perform the matching we must (1) find the smallest value in each row, and (2) sort the row indexes based on the minimum values (**Line 93**). We perform a very similar process on the columns, finding the smallest value in each, and then sorting them based on the ordered rows (**Line 98**). Our goal is to have the index values with the smallest corresponding distance at the front of the lists.

The next step is to use the distances to see if we can associate object IDs:

```
100             # in order to determine if we need to update, register,
101             # or deregister an object we need to keep track of which
102             # of the rows and column indexes we have already examined
103             usedRows = set()
104             usedCols = set()
105
106             # loop over the combination of the (row, column) index
107             # tuples
108             for (row, col) in zip(rows, cols):
109                 # if we have already examined either the row or
110                 # column value before, ignore it
111                 if row in usedRows or col in usedCols:
112                     continue
113
```

```
114                 # if the distance between centroids is greater than
115                 # the maximum distance, do not associate the two
116                 # centroids to the same object
117                 if D[row, col] > self.maxDistance:
118                     continue
119
120                 # otherwise, grab the object ID for the current row,
121                 # set its new centroid, and reset the disappeared
122                 # counter
123                 objectID = objectIDs[row]
124                 self.objects[objectID] = inputCentroids[col]
125                 self.disappeared[objectID] = 0
126
127                 # indicate that we have examined each of the row and
128                 # column indexes, respectively
129                 usedRows.add(row)
130                 usedCols.add(col)
```

Inside the code block, we start by initializing two sets to determine which row and column indexes we have already used (**Lines 103 and 104**). Keep in mind that a set is similar to a list but it contains only *unique* values.

Then we loop over the combinations of `(row, col)` index tuples (**Line 108**) in order to update our object centroids. If we've already used either this row or column index, ignore it and continue to loop (**Lines 111 and 112**). Similarly, if the distance between centroids exceeds the `maxDistance`, then we will not associate the two centroids (**Lines 117 and 118**).

Remark. *This is new functionality compared to my original blog post implementation because we are tracking objects via a background subtraction and centroids rather than an object detector.*

Otherwise, we have found an input centroid that (1) has the smallest Euclidean distance to an existing centroid, and (2) has not been matched with any other object. In that case, we update the object centroid and make sure to add the row and col to their respective `usedRows` and `usedCols` sets (**Lines 123-130**)

There are also likely indexes in our `usedRows` and `usedCols` sets that we have *NOT* examined yet:

```
132             # compute both the row and column index we have NOT yet
133             # examined
134             unusedRows = set(range(0, D.shape[0])).difference(usedRows)
135             unusedCols = set(range(0, D.shape[1])).difference(usedCols)
```

We must determine which centroid indexes we haven't examined yet and store them in two new convenient sets (`unusedRows` and `unusedCols`) on **Lines 134 and 135**.

Our final check handles any objects that have become lost or if they've potentially disappeared:

```
137             # in the event that the number of object centroids is
138             # equal or greater than the number of input centroids
139             # we need to check and see if some of these objects have
140             # potentially disappeared
141             if D.shape[0] >= D.shape[1]:
142                 # loop over the unused row indexes
143                 for row in unusedRows:
144                     # grab the object ID for the corresponding row
145                     # index and increment the disappeared counter
146                     objectID = objectIDs[row]
147                     self.disappeared[objectID] += 1
148
149                     # check to see if the number of consecutive
150                     # frames the object has been marked "disappeared"
151                     # for warrants deregistering the object
152                     if self.disappeared[objectID] > self.maxDisappeared:
153                         self.deregister(objectID)
```

To finish up, if the number of object centroids is greater than or equal to the number of input centroids (**Line 141**), we need to verify if any of these objects are lost or have disappeared by looping over unused row indexes, if any (**Line 143**). In the loop, we will increment their `disappeared` count in the dictionary (**Line 147**). Then we check if the `disappeared` count exceeds the `maxDisappeared` threshold (**Line 152**). If so, we'll `deregister` the object (**Line 153**).

Otherwise, the number of input centroids is greater than the number of existing object centroids, so we have new objects to `register` and track:

```
155             # otherwise, if the number of input centroids is greater
156             # than the number of existing object centroids we need to
157             # register each new input centroid as a trackable object
158             else:
159                 for col in unusedCols:
160                     self.register(inputCentroids[col])
161
162         # return the set of trackable objects
163         return self.objects
```

We loop over the `unusedCols` indexes (**Line 159**) and we register each new centroid (**Line 160**).

Finally, we'll `return` the set of trackable objects to the calling method (**Line 163**).

19.2.3 Trackable Objects

In order to track and count an object in a video stream, we need an easy way to store information regarding the object itself, including:

- It's object ID
- It's previous centroids (so we can easily compute the direction the object is moving)
- Whether or not the object has already been counted

To accomplish all of these goals we can define an instance of `TrackableObject` — open up the `trackableobject.py` file and insert the following code:

```
1  class TrackableObject:
2      def __init__(self, objectID, centroid):
3          # store the object ID, then initialize a list of centroids
4          # using the current centroid
5          self.objectID = objectID
6          self.centroids = [centroid]
7
8          # initialize a boolean used to indicate if the object has
9          # already been counted or not
10         self.counted = False
```

We will have multiple trackable objects — one for each person that is being tracked in the frame. Each object will have the three attributes shown on **Lines 5-10**.

The `TrackableObject` constructor accepts an `objectID` + `centroid` and stores them. The `centroids` variable is a list because it will contain an object's centroid location history. The constructor also initializes `counted` as `False`, indicating that the object has not been counted yet.

19.2.4 The Centroid Tracking Distance Relationship

Recall that each `TrackableObject` has a `centroids` attribute — a list of the object's location history. The algorithm described in the `update` method relies on the history to associate centroids and object IDs by calculating the distance between them.

Calculating the distance between points on a cartesian coordinate system can be done in several ways. The most common method is "as the crow flies", or more formally known as the Euclidean distance.

If you're having trouble following along with **Lines 86-99** open a Python shell and let's practice:

```
1  >>> from scipy.spatial import distance as dist
2  >>> import numpy as np
3  >>> np.random.seed(42)
4  >>> objectCentroids = np.random.uniform(size=(2, 2))
5  >>> centroids = np.random.uniform(size=(3, 2))
6  >>> D = dist.cdist(objectCentroids, centroids)
7  >>> D
8  array([[0.82421549, 0.32755369, 0.33198071],
9         [0.72642889, 0.72506609, 0.17058938]])
```

Once you've started a Python shell in your terminal with the python command, import `distance` and `numpy` as shown on **Lines 1 and 2**).

Then, set a seed for reproducibility (**Line 3**) and generate two (random) existing `objectCentroids` (**Line 4**) and three `inputCentroids` (**Line 5**).

From there, compute the Euclidean distance between the pairs (**Line 6**) and display the results (**Lines 7-9**). The result is a matrix `D` of distances with two rows (# of existing object centroids) and three columns (# of new input centroids).

Just like we did earlier in the script, let's find the minimum distance in each row and sort the indexes based on this value:

```
10 >>> D.min(axis=1)
11 array([0.32755369, 0.17058938])
12 >>> rows = D.min(axis=1).argsort()
13 >>> rows
14 array([1, 0])
```

First, we find the minimum value for each row, allowing us to figure out which existing object is closest to the new input centroid (**Lines 10 and 11**). By then sorting on these values (**Line 12**) we can obtain the indexes of these rows (**Lines 13 and 14**).

In this case, the second row (index 1) has the smallest value and then the first row (index 0) has the next smallest value.

Let's perform a similar process for the columns:

```
15 >>> D.argmin(axis=1)
16 array([1, 2])
17 >>> cols = D.argmin(axis=1)[rows]
18 >>> cols
19 array([2, 1])
```

We first examine the values in the columns and find the index of the value with the smallest

column (**Lines 15 and 16**). We then sort these values using our existing rows (**Lines 17-19**). Let's print the results and analyze them:

```
20 >>> print(list(zip(rows, cols)))
21 [(1, 2), (0, 1)]
```

The final step is to combine them using `zip` (**Lines 20**). The resulting list is printed on **Line 21**.

Analyzing the results, we can make two observations. First, `D[1, 2]` has the smallest Euclidean distance implying that the second existing object will be matched against the third input centroid. And second, `D[0, 1]` has the next smallest Euclidean distance which implies that the first existing object will be matched against the second input centroid.

I'd like to reiterate here that now that you've reviewed the code, you should go back and review the steps to the algorithm in the previous section. From there you'll be able to associate the code with the more linear steps outlined here.

19.2.5 The `DirectionCounter` Class

Another difference (and highly requested feature) between my August 2018 blog post implementation and this book's implementation of people counting is that we can count people from (a) up/down, or (b) left/right. Previously, the app only counted upwards/downwards movement through the frame.

With this new functionality comes a new class, `DirectionCounter`, to manage the task of counting in either vertically or horizontally.

Let's go ahead and build our constructor:

```
1  # import the necessary packages
2  import numpy as np
3
4  class DirectionCounter:
5      def __init__(self, directionMode, H, W):
6          # initialize the height and width of the input image
7          self.H = H
8          self.W = W
9
10         # initialize variables holding the direction of movement,
11         # along with counters for each respective movement (i.e.,
12         # left-to-right and top-to-bottom)
13         self.directionMode = directionMode
14         self.totalUp = 0
15         self.totalDown = 0
```

```
16          self.totalRight = 0
17          self.totalLeft = 0
18
19          # the direction the trackable object is moving in
20          self.direction = ""
```

Our constructor accepts three parameters. We must pass the `directionMode`, either `"vertical"` or `"horizontal"`, that we will be counting our objects.

We also must pass the the height and width of the input image, `H` and `W`.

Instance variables are then initialized. Draw your attention to **Lines 14-17** where the totals are initialized to zero. If our `directionMode` is `"vertical"`, we only care about `totalUp` and `totalDown`. Similarly, if our `directionMode` is `"horizontal"` we are only concerned with `totalRight` and `totalLeft`.

Next, draw your attention to a trackable object's direction `direction` on **Line 21**. We will calculate an object's direction in the the following `find_direction` function:

```
22      def find_direction(self, to, centroid):
23          # check to see if we are tracking horizontal movements
24          if self.directionMode == "horizontal":
25              # the difference between the x-coordinate of the
26              # *current* centroid and the mean of *previous* centroids
27              # will tell us in which direction the object is moving
28              # (negative for 'left' and positive for 'right')
29              x = [c[0] for c in to.centroids]
30              delta = centroid[0] - np.mean(x)
31
32              # determine the sign of the delta -- if it is negative,
33              # the object is moving left
34              if delta < 0:
35                  self.direction = "left"
36
37              # otherwise if the sign is positive, the object is moving
38              # right
39              elif delta > 0:
40                  self.direction = "right"
41
42          # otherwise we are tracking vertical movements
43          elif self.directionMode == "vertical":
44              # the difference between the y-coordinate of the
45              # *current* centroid and the mean of *previous* centroids
46              # will tell us in which direction the object is moving
47              # (negative for 'up' and positive for 'down')
48              y = [c[1] for c in to.centroids]
49              delta = centroid[1] - np.mean(y)
50
51              # determine the sign of the delta -- if it is negative,
52              # the object is moving up
```

```
53              if delta < 0:
54                  self.direction = "up"
55
56              # otherwise, if the sign of the delta is positive, the
57              # object is moving down
58              elif delta > 0:
59                  self.direction = "down"
```

The `find_direction` function accepts a trackable object (`to`) and a single `centroid`. Recall that `centroids` contains a historical listing of an object's position. **Line 30 or 49** grabs all *x*-values or *y*-values from the trackable object's historical centroids, respectively.

The `delta` is calculated by averaging all previous centroid *x* or *y*-coordinate values and subtracting it from the very first value.

From there, if the `delta` is negative, the object is either moving `"left"` or `"down"` (**Lines 34 and 35** plus **Lines 53 and 54**).

Or if the `delta` is positive, the object is either moving `"right"` (**Lines 39 and 40**) or `"up"` (**Lines 58 and 59**).

Now let's implement `count_object`, a function which actually performs the counting:

```
61      def count_object(self, to, centroid):
62          # initialize the output list
63          output = []
64
65          # check if the direction of the movement is horizontal
66          if self.directionMode == "horizontal":
67              # if the object is currently left of center and is
68              # moving left, count the object as moving left
69              leftOfCenter = centroid[0] < self.W // 2
70              if self.direction == "left" and leftOfCenter:
71                  self.totalLeft += 1
72                  to.counted = True
73
74              # otherwise, if the object is right of center and moving
75              # right, count the object as moving right
76              elif self.direction == "right" and not leftOfCenter:
77                  self.totalRight += 1
78                  to.counted = True
79
80              # construct a list of tuples with the count of objects
81              # that have passed in the left and right direction
82              output = [("Left", self.totalLeft),
83                  ("Right",self.totalRight)]
```

The `count_object` function accepts a trackable object and `centroid`. It will create/update a list (`output`) of 2-tuples indicating left/right counts or up/down counts.

Lines 66-83 handle the `"horizontal"` direction mode. If the object is `leftOfCenter` and is moving left, then it is marked as counted and the `totalLeft` is incremented.

Similarly, if the object is `not leftOfCenter` (i.e. right of center) and it is moving `"right"`, then `totalRight` is incremented and the object is marked as counted.

The next code block operates in the exact same fashion, but for the `"vertical"` direction:

```
85          # otherwise the direction of movement is vertical
86          elif self.directionMode == "vertical":
87              # if the the centroid is above the middle and is moving
88              # up, count the object as moving up
89              aboveMiddle = centroid[1] < self.H // 2
90              if self.direction == "up" and aboveMiddle:
91                  self.totalUp += 1
92                  to.counted = True
93
94              # otherwise, if the object is moving down and is below
95              # the middle, count the object as moving down
96              elif self.direction == "down" and not aboveMiddle:
97                  self.totalDown += 1
98                  to.counted = True
99
100             # return a list of tuples with the count of objects that
101             # have passed in the up and down direction
102             output = [("Up", self.totalUp), ("Down", self.totalDown)]
103
104         # return the output list
105         return output
```

This time we're testing if the object is `aboveMiddle` or not and whether it is moving `"up"` or `"down"`. The logic is identical.

In the next section, we will proceed to implement our people counter driver script.

19.2.6 Implementing Our People Counting App Based on Background Subtraction

We have all components/classes ready at this point. Now let's go ahead and implement our people counting driver script.

Go ahead and open a new file named `people_counter.py` and insert the following code:

```
1  # import the necessary packages
2  from pyimagesearch.directioncounter import DirectionCounter
3  from pyimagesearch.centroidtracker import CentroidTracker
4  from pyimagesearch.trackableobject import TrackableObject
```

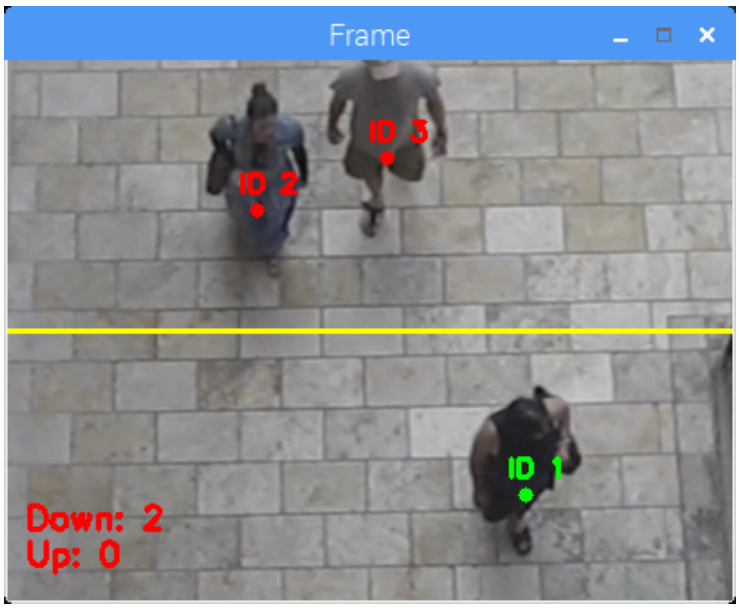

Figure 19.3: Our people counting GUI. The *yellow* counting line will be where objects have to pass to be counted as moving up or down (counts are shown in *red* in the *bottom-left*). Object entroids and object IDs are shown as either *red* (not counted yet) or *green* counted. In this frame, ID #0 (not pictured) has been counted as moving down, ID #1 has been counted as moving down (this person moved down and across the counting line), and IDs #2 and #3 have not yet been counted (they are moving down but have not crossed the counting line).

```
5  from multiprocessing import Process
6  from multiprocessing import Queue
7  from multiprocessing import Value
8  from imutils.video import VideoStream
9  from imutils.video import FPS
10 import argparse
11 import imutils
12 import time
13 import cv2
```

Lines 2-13 import necessary packages. We will use our `DirectionCounter`, `CentroidTracker`, and `TrackableObject`. Taking advantage of `multiprocessing`, we need `Process`, `Queue`, and `Value` for writing to video in a separate process to achieve higher FPS.

Let's define that process for writing to video files now:

```
15 def write_video(outputPath, writeVideo, frameQueue, W, H):
16     # initialize the FourCC and video writer object
17     fourcc = cv2.VideoWriter_fourcc(*"MJPG")
18     writer = cv2.VideoWriter(outputPath, fourcc, 30,
```

```
19          (W, H), True)
20
21      # loop while the write flag is set or the output frame queue is
22      # not empty
23      while writeVideo.value or not frameQueue.empty():
24          # check if the output frame queue is not empty
25          if not frameQueue.empty():
26              # get the frame from the queue and write the frame
27              frame = frameQueue.get()
28              writer.write(frame)
29
30      # release the video writer object
31      writer.release()
```

The `write_video` function will run in an independent process, writing frames from the `frameQueue` to a video file. It accepts five parameters: (1) `outputPath`, the filepath to the output video file, (2) `writeVideo`, a flag indicating if video writing should be ongoing, (3) `frameQueue`, a process-safe `Queue` holding our frames to be written to disk in the video file, and finally, (4 and 5) the video file dimensions.

Lines 18-20 initialize the video `writer`. From there, an infinite loop starts on **Line 24** and will continue to write to the video until `writeVideo` is `False`. The frames are written as they become available in the `frameQueue`.

When the video is finished, it is released (**Line 32**).

With our video writing process out of the way, now let's define our command line arguments:

```
33  # construct the argument parser and parse the arguments
34  ap = argparse.ArgumentParser()
35  ap.add_argument("-m", "--mode", type=str, required=True,
36      choices=["horizontal", "vertical"],
37      help="direction in which people will be moving")
38  ap.add_argument("-i", "--input", type=str,
39      help="path to optional input video file")
40  ap.add_argument("-o", "--output", type=str,
41      help="path to optional output video file")
42  ap.add_argument("-s", "--skip-frames", type=int, default=30,
43      help="# of skip frames between detections")
44  args = vars(ap.parse_args())
```

Our driver script accepts four command line arguments:

- `--mode`: The direction (either `horizontal` or `vertical` in which people will be moving through the frame.
- `--input`: The path to an optional input video file.

- `--output`: The path to an optional output video file. When an output video filepath is provided, the `write_video` process will come to life.
- `--skip-frames`: The number of skip frames used to improve computational efficiency on the RPi. By default we'll skip `30` frames between detections. In between we'll simply correlate and track the detections that have already been made.

Our script can handle webcam or video file streams:

```
46 # if a video path was not supplied, grab a reference to the webcam
47 if not args.get("input", False):
48     print("[INFO] starting video stream...")
49     # vs = VideoStream(src=0).start()
50     vs = VideoStream(usePiCamera=True).start()
51     time.sleep(2.0)
52
53 # otherwise, grab a reference to the video file
54 else:
55     print("[INFO] opening video file...")
56     vs = cv2.VideoCapture(args["input"])
57
58 # initialize the video writing process object (we'll instantiate
59 # later if need be) along with the frame dimensions
60 writerProcess = None
61 W = None
62 H = None
```

Lines 48-52 start a PiCamera video stream. If you'd like to use a USB camera, you can swap the comment symbol between **Lines 50 and 51**. Otherwise, a video file stream will be initialized and started (**Lines 55-57**).

We have a handful more initializations to take care of:

```
63 # instantiate our centroid tracker, then initialize a list to store
64 # each of our dlib correlation trackers, followed by a dictionary to
65 # map each unique object ID to a trackable object
66 ct = CentroidTracker(maxDisappeared=15, maxDistance=100)
67 trackers = []
68 trackableObjects = {}
69
70 # initialize the direction info variable (used to store information
71 # such as up/down or left/right people count)
72 directionInfo = None
73
74 # initialize the MOG foreground background subtractor and start the
75 # frames per second throughput estimator
76 mog = cv2.bgsegm.createBackgroundSubtractorMOG()
77 fps = FPS().start()
```

Our video `writerProcess` placeholder is initialized as `None` on **Line 61** along with the placeholders for the dimensions (**Lines 62 and 63**).

The `directionInfo` is a list of two tuples in the format `[("Up", totalUp), ("Down", totalDown)]` or `[("Left", totalLeft), ("Right", totalRight)]` This variable contains the counts of people that have moved up and down or left and right. **Line 74** initializes the `directionInfo` as `None`.

Our last initializations include our MOG background subtractor (**Line 78**) and our FPS counter (**Line 79).**

Now let's get to work processing frames:

```
80  # loop over frames from the video stream
81  while True:
82      # grab the next frame and handle if we are reading from either
83      # VideoCapture or VideoStream
84      frame = vs.read()
85      frame = frame[1] if args.get("input", False) else frame
86
87      # if we are viewing a video and we did not grab a frame then we
88      # have reached the end of the video
89      if args["input"] is not None and frame is None:
90          break
91
92      # set the frame dimensions and instantiate direction counter
93      # object if required
94      if W is None or H is None:
95          (H, W) = frame.shape[:2]
96          dc = DirectionCounter(args["mode"], H, W)
```

Our `while` loop begins on **Line 82**. A frame is grabbed and indexed depending on if it is from a webcam or video stream (**Lines 86-91**).

If the dimensions of the frame haven't been initialized, it is our signal to initialize them in addition to our `DirectionCounter` (**Lines 95-97**). Notice that we pass the `"mode"` to the direction counter, which will be either `"vertical"` or `"horizontal"`, as well as the frame dimensions.

Next, we'll start our writer process (if we will be writing to a video file):

```
98      # begin writing the video to disk if required
99      if args["output"] is not None and writerProcess is None:
100         # set the value of the write flag (used to communicate when
101         # to stop the process)
102         writeVideo = Value('i', 1)
103
```

```
104         # initialize a shared queue for the exhcange frames,
105         # initialize a process, and start the process
106         frameQueue = Queue()
107         writerProcess = Process(target=write_video, args=(
108             args["output"], writeVideo, frameQueue, W, H))
109         writerProcess.start()
```

If we have an `"output"` video path in `args` and the `writerProcess` doesn't yet exist (**Line 100**), then **Lines 103-110** set the `writeVideo` flag, initialize our `frameQueue`, and `start` our `writerProcess`.

Moving on, let's preprocess our frame and **apply background subtraction:**

```
111     # initialize a list to store the bounding box rectangles returned
112     # by background subtraction model
113     rects = []
114
115     # convert the frame to grayscale and smoothen it using a
116     # gaussian kernel
117     gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
118     gray = cv2.GaussianBlur(gray, (5, 5), 0)
119
120     # apply the MOG background subtraction model
121     mask = mog.apply(gray)
122
123     # apply a series of erosions to break apart connected
124     # components, then find contours in the mask
125     erode = cv2.erode(mask, (7, 7), iterations=2)
126     cnts = cv2.findContours(erode.copy(), cv2.RETR_EXTERNAL,
127         cv2.CHAIN_APPROX_SIMPLE)
128     cnts = imutils.grab_contours(cnts)
```

Line 114 initializes an empty list to store bounding box rectangles which will be returned by the background subtraction model.

Lines 118 and 119 convert the frame to grayscale and blur it slightly. Background subtraction is applied to the `gray` frame (**Line 122**). From there, a series of erosions are applied to reduce noise (**Line 126**).

Then contours are found and extracted via **Lines 127-129**. Let's process our contours and add the bounding boxes to `rects`:

```
130     # loop over each contour
131     for c in cnts:
132         # if the contour area is less than the minimum area
133         # required then ignore the object
134         if cv2.contourArea(c) < 2000:
```

```
135             continue
136
137         # compute the bounding box coordinates of the contour
138         (x, y, w, h) = cv2.boundingRect(c)
139         (startX, startY, endX, endY) = (x, y, x + w, y + h)
140
141         # add the bounding box coordinates to the rectangles list
142         rects.append((startX, startY, endX, endY))
```

For each contour, if it is sufficiently large, we extract its bounding rectangle coordinates (**Lines 132-140**. Then we add it to the `rects` list (**Line 143**).

Now we will split the screen with either a vertical or horizontal line:

```
144     # check if the direction is vertical
145     if args["mode"] == "vertical":
146         # draw a horizontal line in the center of the frame -- once an
147         # object crosses this line we will determine whether they were
148         # moving 'up' or 'down'
149         cv2.line(frame, (0, H // 2), (W, H // 2), (0, 255, 255), 2)
150
151     # otherwise, the direction is horizontal
152     else:
153         # draw a vertical line in the center of the frame -- once an
154         # object crosses this line we will determine whether they
155         # were moving 'left' or 'right'
156         cv2.line(frame, (W // 2, 0), (W // 2, H), (0, 255, 255), 2)
```

If our `"mode"` is `"vertical"`, we draw a horizontal line in the middle of the screen (**Lines 146-150**). Otherwise, if our `"mode"` is `"horizontal"`, we draw a vertical line in the center of the screen (**Lines 153-157**).

Either line will serve as a visual indication of the point at which a person must pass to be counted.

Let's count our objects — this is the heart of the driver script:

```
158     # use the centroid tracker to associate the (1) old object
159     # centroids with (2) the newly computed object centroids
160     objects = ct.update(rects)
161
162     # loop over the tracked objects
163     for (objectID, centroid) in objects.items():
164         # grab the trackable object via its object ID
165         to = trackableObjects.get(objectID, None)
166         color = (0, 0, 255)
167
168         # create a new trackable object if needed
```

```
169             if to is None:
170                 to = TrackableObject(objectID, centroid)
171
172             # otherwise, there is a trackable object so we can utilize it
173             # to determine direction
174             else:
175                 # find the direction and update the list of centroids
176                 dc.find_direction(to, centroid)
177                 to.centroids.append(centroid)
178
179                 # check to see if the object has been counted or not
180                 if not to.counted:
181                     # find the direction of motion of the people
182                     directionInfo = dc.count_object(to, centroid)
183
184                 # otherwise, the object has been counted and set the
185                 # color to green indicate it has been counted
186                 else:
187                     color = (0, 255, 0)
```

Line 160 calls `update` on our centroid tracker to associate old object centroids with the freshly computed centroids. Under the hood, this is where **Steps #2 through #5** from Section 19.2.2.2 from the previous section take place.

From there, we'll loop over the centroid `objects` beginning on **Line 163**.

The goals of this loop includes (1) tracking objects, (2) determining the direction the objects are moving, and (3) counting the objects depending on their direction of motion.

Line 165 grabs a trackable object via its ID.

Line 166 sets the default centroid + ID `color` to *red* for now. It will soon become *green* as the person becomes counted.

Lines 169 and 170 create a new trackable object if needed.

Otherwise, the trackable object's direction is determined from its centroid history using the `DirectionCounter` class (**Line 176**). The centroid history is updated via **Line 177**. Finally, if the trackable object hasn't been counted yet (**Line 180**), it is counted (**Line 182**). Otherwise, it has already been counted and we must ensure that its `color` is *green* as it has passed the counting line (**Lines 186 and 187**). **Line 190** stores the trackable object in our `trackableObjects` dictionary.

Be sure to refer to the `CentroidTracker`, `TrackableObject`, and `DirectionCounter` classes at this point to see what is going on under the hood in **Lines 160-190.**

The remaining three codeblocks are for display/video annotation and housekeeping. Let's annotate our frame:

```
189         # store the trackable object in our dictionary
190         trackableObjects[objectID] = to
191
192         # draw both the ID of the object and the centroid of the
193         # object on the output frame
194         text = "ID {}".format(objectID)
195         cv2.putText(frame, text, (centroid[0] - 10,     centroid[1] - 10),
196             cv2.FONT_HERSHEY_SIMPLEX, 0.5, color, 2)
197         cv2.circle(frame, (centroid[0], centroid[1]), 4, color, -1)
```

Lines 190-197 annotate each object with a small dot and an ID number.

Now let's annotate the object counts in the corner:

```
199     # extract the people counts and write/draw them
200     if directionInfo is not None:#
201         for (i, (k, v)) in enumerate(directionInfo):
202             text = "{}: {}".format(k, v)
203             cv2.putText(frame, text, (10, H - ((i * 20) + 20)),
204                 cv2.FONT_HERSHEY_SIMPLEX, 0.6, (0, 0, 255), 2)
```

Using the `directionInfo`, we extract the up/down or left/right object counts. The text is displayed in the corner of the frame.

From here we'll finish out our loop and then clean up:

```
206     # put frame into the shared queue for video writing
207     if writerProcess is not None:#
208         frameQueue.put(frame)
209
210     # show the output frame
211     cv2.imshow("Frame", frame)
212     key = cv2.waitKey(1) & 0xFF
213
214     # if the `q` key was pressed, break from the loop
215     if key == ord("q"):
216         break
217
218     # update the FPS counter
219     fps.update()
220
221 # stop the timer and display FPS information
222 fps.stop()
223 print("[INFO] elapsed time: {:.2f}".format(fps.elapsed()))
224 print("[INFO] approx. FPS: {:.2f}".format(fps.fps()))
225
226 # terminate the video writer process
227 if writerProcess is not None:
```

```
228         writeVideo.value = 0
229         writerProcess.join()
230
231     # if we are not using a video file, stop the camera video stream
232     if not args.get("input", False):
233         vs.stop()
234
235     # otherwise, release the video file pointer
236     else:
237         vs.release()
238
239     # close any open windows
240     cv2.destroyAllWindows()
```

Lines 207 and 208 add an annotated frame to the `frameQueue`. The `writerProcess` will process the frames in the background using multiprocessing. The frame is shown on the screen until the `q` (quit) keypress is detected (**Lines 211-216**). The FPS counter is updated via **Line 219**. Assuming we've broken out of the `while` loop, **Lines 222-240** perform cleanup and print FPS statistics.

In the next section, we'll deploy our people counter!

19.2.7 Deploying the RPi People Counter

In this chapter so far, we've reviewed the mechanics of tracking and counting objects. Then we brought it all together with our driver script. We've now reached the fun part — putting it to the test.

Remember, this people counter differs from the one presented on my blog (http://pyimg.co/vgak6) [72]. This script is optimized such that the Raspberry Pi can successfully count people.

When you're ready, go ahead and execute the following command using the input video to test with:

```
$ python people_counter.py --mode vertical \
    --input videos/veritcal_01.mp4 \
    --output output/vertical_01.avi
```

Figure 19.4 shows three separate frames from people counter processing. There are no issues with the *left* and *center* frames. However, on the *right* two people are only being tracked as one centroid (ID #9). Obviously, this results in a counting error.

But is there a way around it?

Of course you could tune your morphological operations (i.e. the erosion and dilation kernels). Doing so would fix the problem in this case, but not under all scenarios.

Figure 19.4: Examples of people counting using a background subtraction + centroid tracking method. On the *right*, notice that two contours blobs were touching (ID #9), indicating that only one centroid is tracked; this results in a counting error and shows the disadvantage of this chapter's methodology. A more accurate object detector (HOG, SSD) would have resulted in both of these people being independent objects (covered in the *Hacker Bundle*).

Does this limit us? Are there alternatives?

Yes, but with a severely resource-constrained device, it seems that we have no choice but to rely on a pipeline similar to the one discussed in this chapter (background subtraction with centroid tracking). An alternative, assuming you have the processing horsepower, would be to rely on an object detection model (HOG or SSD) every *N* frames which will better discern the difference between separate people. We add the horsepower to a RPi (a Movidius Neural Compute Stick) and will learn such a method in the *Hacker Bundle*.

Our RPi 3B+ achieved **19.65 FPS** when benchmarking this project. Our goal was essentially 8FPS or higher to ensure that we counted each walking person, so we did well on the pipeline speed.

The RPi 4 (1GB version) achieved a whopping **35.74 FPS**. Taking it even further, the RPi 4 (4GB version) achieved an ***insanely high* 50.06 FPS**. As you can see, the RPi 4 is more than than 2x faster than the RPi 3B+.

With a live webcam, FPS will certainly help with accuracy such as during a running race while people are moving faster. The more often centroids are correlated via the `CentroidTracker`, the better.

You can run the code with other command line arguments to suit your needs. To run the people counting app on a live video feed, be sure to omit the `--input` command line argument (you can leave off the `--output` argument if you don't want to save the likely large video file to disk either):

```
$ python people_counter.py --mode vertical
```

Or, if you want to to count people moving horizontally, use a live webcam feed, and output

the resulting video to disk, be sure to change the `--mode` accordingly:

```
$ python people_counter.py --mode horizontal
    --output output/webcam_output.avi
```

19.2.8 Tips and Suggestions

When you mount your people counter in a public/private place, be sure you follow all surveillance laws in your area.

Using the background subtraction method, you will need to tune the minimum contour size (**Line 135** of the driver script). The size you choose will be dependent upon how far the camera is from the people and the type of lens you use. Furthermore any frame resizing you perform to speed up your pipeline will also impact contour sizes. The only way to determine this value is by experimentation.

For the example video used during development, only erosion morphological operations were applied to the background subtraction mask. You may need to add dilation. Additionally, you may need to adjust the kernel size and number of iterations. This will be dependent upon noise in the image (there shouldn't be too much for a static ground) and the size of the objects/frame.

Finally, it may be the case that simple background subtraction is not sufficient for your particular task. Background subtraction, while efficient (especially on resource constrained devices such as the RPi), is still a basic algorithm that has *no semantic understanding* of the video stream itself.

Instead, you may need to leverage dedicated object detection algorithms, including Haar cascades, HOG + Linear SVM, and deep learning-based detectors. We'll be covering how to utilize these more advanced methods inside the *Hacker Bundle* of this text.

19.3 Summary

In this chapter, we designed a people counter based on a number of concepts.

Background subtraction and contours were used to find moving objects (i.e. people). We implemented a Haar Cascade object detector to locate people.

We developed a `CentroidTracker` to maintain a lookup table for objects in the frame. This class was responsible for registering/deregistering objects, and updating their locations. Our `TrackableObject` class keeps track of a person's ID, current location, and historical locations. From there, the `DirectionCounter` class counted left/right (horizontal) or up/down

(vertical) moving people depending on the set mode.

Given that this chapter spans many pages including three new classes and a driver script, it is easy to become lost. If you're experiencing this feeling, then I suggest you open up the project folder in your editor/IDE and study it on your screen. Then open up the book and review the chapter once again.

In the next chapter, we're going to apply what we learned here to traffic counting. The concepts are the same, but there a selection of additional considerations we have to make for implementation.

This chapter on *“Building a Smart Attendance System”* is from the *Hacker Bundle*…

Chapter 6

Building a Smart Attendance System

In this chapter you will learn how to build a smart attendance system used in school and classroom applications.

Using this system, you, or a teacher/school administrator, can take attendance for a classroom using *only* face recognition — no manual intervention of the instructor is required.

To build such an application, we'll be using computer vision algorithms and concepts we've learned throughout the text, including accessing our video stream, detecting faces, extracting face embeddings, training a face recognizer, and then finally putting all the components together to create the final smart classroom application.

Since this chapter references techniques used in so many previous chapters, I *highly recommend* that you read all preceding chapters in the book before you proceed.

6.1 Chapter Learning Objectives

In this chapter you will:

i. Learn about smart attendance applications and why they are useful

ii. Discover TinyDB and how it can be used in real-world applications

iii. Implement and perform face enrollment/registration

iv. Train a face recognition model

v. Implement the final smart classroom application

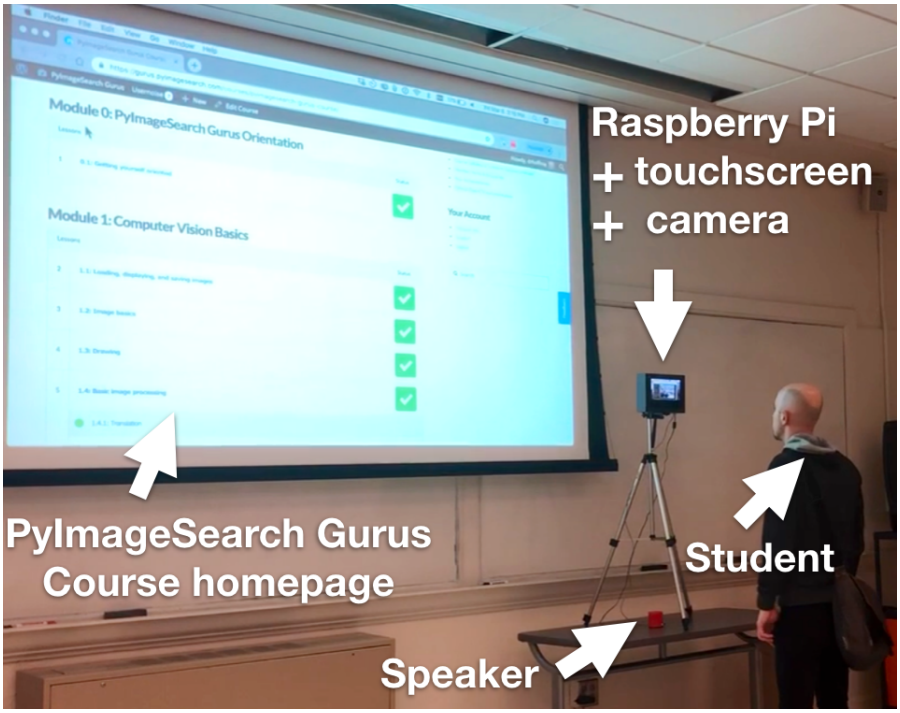

Figure 6.1: You can envision your attendance system being placed near where students enter the classroom at the front doorway. You will need a screen with adjacent camera along with a speaker for audible alerts.

6.2 Overview of Our Smart Attendance System

We'll start this section with a brief review of what a smart attendance application is and why we may want to implement our own smart attendance system.

From there we'll review the directory structure for the project and review the configuration file.

6.2.1 What is a Smart Attendance System?

The goal of a smart attendance system is to *automatically* recognize students and take attendance *without* having the instructor having to manually intervene. Freeing the instructor from having to take attendance gives the teacher more time to interact with the students and do what they do best — *teach* rather than *administer.*

An example of a working smart attendance system can be seen in Figures **??** and 6.2. Notice how as the student walks into a classroom they are automatically recognized. This

positive recognition is then logged to a database, marking the student as “present” for the given session.

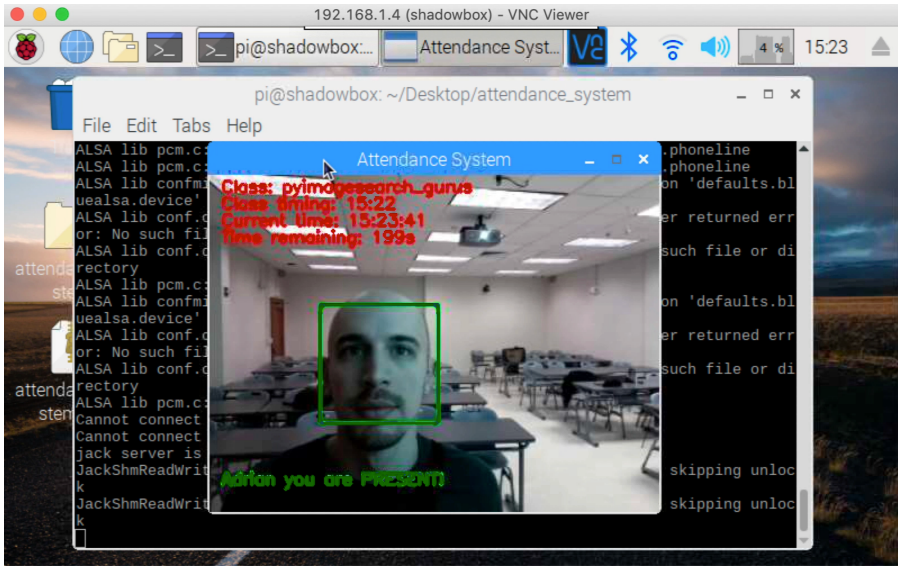

Figure 6.2: An example of a smart attendance system in action. Face detection is performed to detect the presence of a face. Next, the detected face is recognized. Once the person is identified the student is logged as "present" in the database.

We’ll be building our own smart attendance system in the remainder of this chapter.

The application will have multiple steps and components, each detailed below:

i. **Step #1:** Initialize the database (only needs to be done *once*)

ii. **Step #2:** Face enrollment (needs to be performed for each student in the class)

iii. **Step #3:** Train the face recognition model (needs to be performed *once*, and then *again* if a student is ever enrolled or un-enrolled).

iv. **Step #4:** Take attendance (once per classroom session).

Before we start implementing these steps, let’s first review our directory structure for the project.

6.2.2 Project Structure

Let’s go ahead and review our directory structure for the project:

```
|-- config
|   |-- config.json
|-- database
|   |-- attendance.json
|-- dataset
|   |-- pyimagesearch_gurus
|       |-- S1901
|       |   |-- 00000.png
...
|       |   |-- 00009.png
|       |-- S1902
|           |-- 00000.png
...
|           |-- 00009.png
|-- output
|   |-- encodings.pickle
|   |-- le.pickle
|   |-- recognizer.pickle
|-- pyimagesearch
|   |-- utils
|   |   |-- __init__.py
|   |   |-- conf.py
|   |-- __init__.py
|-- initialize_database.py
|-- enroll.py
|-- unenroll.py
|-- encode_faces.py
|-- train_model.py
|-- attendance.py
```

The `config/` directory will store our `config.json` configurations for the project.

The `database/` directory will store our `attendance.json` file which is the serialized JSON output from TinyDB, the database we'll be using for this project.

The `dataset/` directory (not to be confused with the `database/` folder) will store all example faces of each student captured via the `enroll.py` script.

We'll then train a face recognition model on these captured faces via both `encode_faces.py` and `train_model.py` — the output of these scripts will be stored in the `output/` directory.

Our `pyimagesearch` module is quite simplistic, requiring only our `Conf` class used to load our configuration file from disk.

Before building our smart attendance system we must first initialize our database via the `initialize_database.py` script.

Once we have captured example faces for each student, extracted face embeddings, and then trained our face recognition model, we can use `attendance.py` to take attendance.

This is the final script meant to be run in the actual *classroom*. It takes all of the individual components implemented in the project and combines it into the final smart attendance system.

If a student ever needs to leave the class (such as them dropping out of the course), we can run `unenroll.py`.

6.2.3 Our Configuration File

Let's now review our `config.json` file in the `config` directory:

```
1  {
2      // text to speech engine language and speech rate
3      "language": "english-us",
4      "rate": 175,
5
6      // path to the dataset directory
7      "dataset_path": "dataset",
8
9      // school/university code for the class
10     "class": "pyimagesearch_gurus",
11
12     // timing of the class
13     "timing": "14:05",
```

Lines 3 and 4 define the language and rate of speech for our Text-to-Speech (TTS) engine. We'll be using the `pyttsx3` library in this project — if you need to change the `"language"` you can refer to the documentation for your specific language value [CITE].

The `"dataset_path"` points to the `"dataset"` directory. Inside this directory we'll store all captured face ROIs via the `enroll.py` script. We'll then later read all images from this directory and train a face recognition model on top of the faces via `encode_faces.py` and `train_model.py`.

Line 10 sets the `class_name`, which is as the name suggests, is the title of the class. We're calling this class `"pyimagesearch_gurus"`.

We then have the `"timing"` of the class (**Line 13**). This value is the time of day the class actually starts. Our `attendance.py` script will monitor the time and ensure that attendance can only be taken in a *N* second window once class actually starts.

Our next set of configurations for face detection and face recognition:

```
15     // number of face detections before actually saving the faces to
16     // disk
17     "n_face_detection": 10,
```

```
18
19      // number of images required per person in the dataset
20      "face_count": 10,
21
22      // maximum time limit (in seconds) to take the attendance once
23      // the class has started
24      "max_time_limit": 300,
25
26      // number of consecutive recognitions to be made to mark a person
27      // recognized
28      "consec_count": 3,
```

The `"n_face_detection"` value controls the number of subsequent frames with a face detected *before* we save the face ROI to disk. Enforcing at least ten consecutive frames with a face detected prior to saving the face ROI to disk helps reduce false-positive detections.

The `"face_count"` parameter sets the minimum number of face examples per student. Here we are requiring that we capture ten total face examples per student.

We then have `"max_time_limit` — this value sets the maximum time limit (in second) to take attendance for once the class has started. Here we have a value of `300` seconds (five minutes). Once class starts, the students have a total of five minutes to make their way to the classroom, verify that they are present with our smart attendance system, and take their seats (otherwise they will be marked as “absent”).

To reduce misrecognizing a face, it must be recognized `"consec_count"` times in a row. This ensures that if our face recognizer is "flickering" between two names it will take the name that is recognized as "Adrian" multiple times in succession to count as "Adrian" even if it incorrectly reported "Abhishek" for a reading.

Our final code block handles setting file paths:

```
30      // path to the database
31      "db_path": "database/attendance.json",
32
33      // paths to the encodings, recognizer, and label encoder
34      "encodings_path": "output/encodings.pickle",
35      "recognizer_path": "output/recognizer.pickle",
36      "le_path": "output/le.pickle",
37
38      // dlib face detection to be used
39      "detection_method": "hog"
40  }
```

Line 31 defines the `"db_path"` which is the output path to our serialized TinyDB file.

Lines 34-36 set additional file paths:

- `"encodings_path"`: The path to the output 128-d extracted embeddings/quantifications for each face from the `dataset/` directory.
- `"recognizer_path"`: The path to the trained SVM from `train_model.py`.
- `"le_path"`: The serialized `LabelEncoder` object uses to transform predictions to human-readable labels (in this case, the *names* of the students).

Finally, **Line 39** sets our `"detection_method"`. We'll be using the HOG + Linear SVM detector from the `dlib` and `face_recognition` library. Haar cascades would be faster than HOG + Linear SVM, but less accurate. Similarly, deep learning-based face detectors would be *much* more accurate but *far too slow* to run in real-time (especially since we're not only performing face *detection* but face *recognition* as well).

If you are using a co-processor such as the Movidius NCS or Google Coral USB Accelerator I would suggest using the deep learning face detector (as it will be more accurate), but if you're using *just* the Raspberry Pi CPU, stick with either HOG + Linear SVM or Haar cascades as these methods are fast enough to run in (near) real-time on the RPi.

6.3 Step #1: Creating our Database

Before we can enroll faces in our system and take attendance, we first need to initialize the database that will store information on the **class** (name of the class, date/time class starts, etc.) and the **students** (student ID, name, etc.).

We'll be using TinyDB [34] for all database operations. TinyDB is small, efficient, and implemented in pure Python. We're using TinyDB for this project as it allows for database operations to "get out of way", ensuring we can focus on implementing the actual *computer vision algorithms* rather than CRUD operations.

6.3.1 What is TinyDB?

TinyDB is a "tiny, documented oriented database" (http://pyimg.co/eowhr) [34].

The library is written in pure Python, is simple to use, and allows us to store any object represented as a Python `dict` data type.

For example, the following code snippet loads a serialized database from disk, inserts a record for a student, and then demonstrates how to query for that record:

```
>>> from tinydb import TinyDB
>>> from tinydb import Query
```

```
>>> db = TinyDB("path/to/db.json")
>>> Student = Query()
>>> db.insert({"name": "Adrian", "age": 30})
>>> db.search(User.name == "Adrian")
[{"name": "John", "age": 30}]
```

As you can see, TinyDB allows us to focus *less* on the actual database code and *more* on the embedded computer vision/deep learning concepts (which is what this book is about, after all).

If you do not already have the `tinydb` Python package installed on your system, you can install it via the following command:

```
$ pip install tinydb
```

I would recommend you stick with TinyDB to build your own proof-of-concept smart attendance system. Once you're happy with it the system you can then try more advanced, feature-rich databases including mySQL, postgresql, MongoDB, Firebase, etc.

6.3.2 Our Database Schema

Internally, TinyDB represents a database as Python dictionary. Data is stored in (nested) Python dictionaries. When presented with a query, TinyDB scans the Python dictionary objects and returns all items that match the query parameters.

Our database will have three top-level dictionaries:

```
{
  "_default": {
    "1": {
      "class": "pyimagesearch_gurus"
    }
  },
  "attendance": {
    "2": {
      "2019-11-13": {
        "S1901": "08:01:15",
        "S1903": "08:03:41",
        "S1905": "08:04:19"
      }
    },
    "1": {
      "2019-11-14": {
        "S1904": "08:02:22",
        "S1902": "08:02:49",
```

```
19            "S1901": "08:04:27"
20          }
21        }
22      },
23      "student": {
24        "1": {
25          "S1901": [
26            "Adrian",
27            "enrolled"
28          ]
29        },
30        "2": {
31          "S1902": [
32            "David",
33            "enrolled"
34          ]
35        },
36        "3": {
37          "S1903": [
38            "Dave",
39            "enrolled"
40          ]
41        },
42        "4": {
43          "S1904": [
44            "Abhishek",
45            "enrolled"
46          ]
47        },
48        "5": {
49          "S1905": [
50            "Sayak",
51            "enrolled"
52          ]
53        }
54      }
55  }
```

The `class` key contains the name of the class where our smart attendance system will be running. Here you can see that the name of the class, for this example, is *"pyimagesearch_gurus"*.

The `student` dictionary stores information for all students in the database. Each student must have a name and a status flag used to indicate if they are enrolled or un-enrolled in a given class. The actual student ID can be whatever you want, but I've chosen the format:

- `S`: Indicating "student"
- `19`: The current year.
- `01`: The first student to be registered.

The next student registered would then be `S1902`, and so on. You can choose to keep this same ID structure or define your own — the actual ID is entirely arbitrary provided that the ID is *unique*.

Finally, the `attendance` dictionary stores the attendance record for each session of the class. For each session, we store both (1) the **student ID** for each student who attended the class, along with (2) the **timestamp** of when each student was successfully recognized. By recording both of these values we can then determine *which* students attended a class and whether or not they were *late for class*.

Keep in mind this database schema is meant to be the *bare minimum* of what's required to build a smart attendance application. Feel free to add in additional information on the student, including age, address, emergency contact, etc.

Secondly, we're using TinyDB here out of simplicity. When building your own smart attendance application you may wish to use another database — I'll leave that as an exercise to you to implement as the point of this text is to focus on computer vision algorithms rather than database operations.

6.3.3 Implementing the Initialization Script

Our first script, `initialize_database.py`, is a utility script used to create our initial TinyDB database. This script only needs to be executed *once* but it *has to be executed* before you start enrolling faces.

Let's take a look at the script now:

```
# import the necessary packages
from pyimagesearch.utils import Conf
from tinydb import TinyDB
import argparse

# construct the argument parser and parse the arguments
ap = argparse.ArgumentParser()
ap.add_argument("-c", "--conf", required=True,
    help="Path to the input configuration file")
args = vars(ap.parse_args())

# load the configuration file
conf = Conf(args["conf"])

# initialize the database
db = TinyDB(conf["db_path"])

# insert the details regarding the class
print("[INFO] initializing the database...")
db.insert({"class": conf["class"]})
```

```
21 print("[INFO] database initialized...")
22
23 # close the database
24 db.close()
```

Lines 2-4 import our required Python packages. **Line 3** imports the `TinyDB` class used to interface with our database.

Only only command line argument, `--conf`, is parsed on **Lines 7-10**. We then load the `conf` on **Line 13** and use the `"db_path"` to initialize the `TinyDB` instance.

Once we have the `db` object instantiated we use the `insert` method to add data to the database. Here we are adding the `class` name from the configuration file. We need to insert a class so that students can be enrolled in the class in Section 6.4.2.

Finally, we close the database on **Line 24** which triggers the TinyDB library to serialize the database back out to disk as a JSON file.

6.3.4 Initializing the Database

Let's initialize our database now.

Open up a terminal and issue the following command:

```
$ python initialize_database.py --conf config/config.json
[INFO] initializing the database...
[INFO] database initialized...
```

If you check the `database/` directory you'll now see a file named `attendance.json`:

```
$ ls database/
attendance.json
```

The `attendance.json` file is our actual TinyDB database. The TinyDB library will read, manipulate, and save the data inside this file.

6.4 Step #2: Enrolling Faces in the System

Now that we have our database initialized we can move on to face enrollment and un-enrollment.

During **enrollment** a student will stand in front of a camera. Our system will access the camera, perform face detection, extract the ROI of the face and then serialize the ROI to disk.

In Section 6.5 we'll take these ROIs, extract face embeddings, and then train a SVM on top of the embeddings.

A teacher or school administrator can perform **un-enrollment** as well. We may want to un-enroll a student if they leave a class or when the semester ends.

6.4.1 Implementing Face Enrollment

Before we can recognize students in a classroom we first need to "enroll" them in our face recognition system. Enrollment is a two step process:

i. **Step #1:** Capture faces of each individual and record them in our database (covered in this section).

ii. **Step #2:** Train a machine learning model to recognize each individual (covered in Section 6.5).

The first phase of face enrollment will be accomplished via the `enroll.py` script. Open up that script now and insert the following code:

```
1  # import the necessary packages
2  from pyimagesearch.utils import Conf
3  from imutils.video import VideoStream
4  from tinydb import TinyDB
5  from tinydb import where
6  import face_recognition
7  import argparse
8  import imutils
9  import pyttsx3
10 import time
11 import cv2
12 import os
```

Lines 2-12 handle import our required Python packages. The `tinydb` imports on **Lines 4 and 5** will interface with our database. The `where` function will be used to perform SQL-like *"where"* clauses to search our database. **Line 6** imports the `face_recognition` library which will be used to facilitate face detection (in this section) and face recognition (in Section 6.6). The `pyttsx3` import is our Text-to-Speech (TTS) library. We'll be using this package whenever we need to generate speech and play it through our speakers.

Next, let's parse our command line arguments:

```
14 # construct the argument parser and parse the arguments
15 ap = argparse.ArgumentParser()
```

```
16 ap.add_argument("-i", "--id", required=True,
17     help="Unique student ID of the student")
18 ap.add_argument("-n", "--name", required=True,
19     help="Name of the student")
20 ap.add_argument("-c", "--conf", required=True,
21     help="Path to the input configuration file")
22 args = vars(ap.parse_args())
```

We require three command line arguments:

- `--id`: The unique ID of the student.
- `--name`: The name of the student.
- `--conf`: The path to our configuration file.

Let's use TinyDB and query if a student with the given `--id` *already* exists in our database:

```
24 # load the configuration file
25 conf = Conf(args["conf"])
26
27 # initialize the database and student table objects
28 db = TinyDB(conf["db_path"])
29 studentTable = db.table("student")
30
31 # retrieve student details from the database
32 student = studentTable.search(where(args["id"]))
```

Line 25 loads our configuration file from disk.

Using this configuration we then load the TinyDB and grab a reference to the `student` table (**Line 29**).

The `where` method is used to search the `studentTable` for all records which have the supplied `--id`.

If there are *no existing records* with the supplied ID then we know the student *has not been enrolled yet*:

```
34 # check if an entry for the student id does *not* exist, if so, then
35 # enroll the student
36 if len(student) == 0:
37     # initialize the video stream and allow the camera sensor to warmup
38     print("[INFO] warming up camera...")
39     # vs = VideoStream(src=0).start()
40     vs = VideoStream(usePiCamera=True).start()
```

```
41      time.sleep(2.0)
42
43      # initialize the number of face detections and the total number
44      # of images saved to disk
45      faceCount = 0
46      total = 0
47
48      # initialize the text-to-speech engine, set the speech language, and
49      # the speech rate
50      ttsEngine = pyttsx3.init()
51      ttsEngine.setProperty("voice", conf["language"])
52      ttsEngine.setProperty("rate", conf["rate"])
53
54      # ask the student to stand in front of the camera
55      ttsEngine.say("{} please stand in front of the camera until you" \
56          "receive further instructions".format(args["name"]))
57      ttsEngine.runAndWait()
```

Line 36 makes a check to ensure that no existing students have the same `--id` we are using. Provided that check passes we access our video stream and initialize two integers:

- `faceCount`: The number of consecutive frames with a face detected.
- `total`: The total number of faces saved for the current student.

Lines 50-52 initialize the TTS engine by setting the speech language and the speech rate.

We then instruct the student (via the TTS engine) to stand in front of the camera (**Lines 55-57**). With the student now in front of the camera we can capturing faces of the individual:

```
59      # initialize the status as detecting
60      status = "detecting"
61
62      # create the directory to store the student's data
63      os.makedirs(os.path.join(conf["dataset_path"], conf["class"],
64          args["id"]), exist_ok=True)
65
66      # loop over the frames from the video stream
67      while True:
68          # grab the frame from the threaded video stream, resize it (so
69          # face detection will run faster), flip it horizontally, and
70          # finally clone the frame (just in case we want to write the
71          # frame to disk later)
72          frame = vs.read()
73          frame = imutils.resize(frame, width=400)
74          frame = cv2.flip(frame, 1)
75          orig = frame.copy()
```

Line 60 sets our current `status` to `detecting`. Later this status will be updated to `saving` once we start writing example face ROIs to disk.

Lines 63 and 64 create a subdirectory in our `dataset/` directory. This subdirectory is named based on the supplied `--id`.

We then start looping over frames of our video stream on **Line 67**. We preprocess the `frame` by resizing it to have a width of 400px (for faster processing) and then horizontally flipping it (to remove the mirror effect).

Let's now perform face detection on the `frame`:

```
77          # convert the frame from from RGB (OpenCV ordering) to dlib
78          # ordering (RGB) and detect the (x, y)-coordinates of the
79          # bounding boxes corresponding to each face in the input image
80          rgb = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)
81          boxes = face_recognition.face_locations(rgb,
82              model=conf["detection_method"])
83
84          # loop over the face detections
85          for (top, right, bottom, left) in boxes:
86              # draw the face detections on the frame
87              cv2.rectangle(frame, (left, top), (right, bottom),
88                  (0, 255, 0), 2)
89
90              # check if the total number of face detections are less
91              # than the threshold, if so, then skip the iteration
92              if faceCount < conf["n_face_detection"]:
93                  # increment the detected face count and set the
94                  # status as detecting face
95                  faceCount += 1
96                  status = "detecting"
97                  continue
98
99              # save the frame to correct path and increment the total
100             # number of images saved
101             p = os.path.join(conf["dataset_path"], conf["class"],
102                 args["id"], "{}.png".format(str(total).zfill(5)))
103             cv2.imwrite(p, orig[top:bottom, left:right])
104             total += 1
105
106             # set the status as saving frame
107             status = "saving"
```

We use the `face_recognition` library to perform face detection using the HOG + Linear SVM method on **Lines 81 and 82**.

The `face_locations` function returns a list of four values: the top, right, bottom, and left *(x, y)*-coordinates of each face in the image.

On **Line 85** we loop over the detected `boxes` and use the `cv2.rectangle` function to draw the bounding box of the face.

Line 92 makes a check to see if the `faceCount` is still below the number of required consecutive frames with a face detected (used to reduce false-positive detections). If our `faceCount` is below the threshold we increment the counter and `continue` looping.

Once we have reached the threshold we derive the path to the output face ROI (**Lines 101 and 102**) and then write the face ROI to disk (**Line 103**). We then increment our `total` face ROI count and update the `status`.

We can then draw the `status` on the `frame` and visualize it on our screen:

```
109         # draw the status on to the frame
110         cv2.putText(frame, "Status: {}".format(status), (10, 20),
111             cv2.FONT_HERSHEY_SIMPLEX, 0.5, (255, 0, 0), 2)
112
113         # show the output frame
114         cv2.imshow("Frame", frame)
115         cv2.waitKey(1)
116
117         # if the required number of faces are saved then break out from
118         # the loop
119         if total == conf["face_count"]:
120             # let the student know that face enrolling is over
121             ttsEngine.say("Thank you {} you are now enrolled in the {} " \
122                 "class.".format(args["name"], conf["class"]))
123             ttsEngine.runAndWait()
124             break
125
126     # insert the student details into the database
127     studentTable.insert({args["id"]: [args["name"], "enrolled"]})
128
129     # print the total faces saved and do a bit of cleanup
130     print("[INFO] {} face images stored".format(total))
131     print("[INFO] cleaning up...")
132     cv2.destroyAllWindows()
133     vs.stop()
134
135 # otherwise, a entry for the student id exists
136 else:
137     # get the name of the student
138     name = student[0][args["id"]][0]
139     print("[INFO] {} has already already been enrolled...".format(
140         name))
141
142 # close the database
143 db.close()
```

If our `total` reaches the maximum number of `face_count` images needed to train our

face recognition model (**Line 119**), we use the TTS engine to tell the user enrollment for them is now complete (**Lines 121-123)**. The student is then inserted into the TinyDB, including the ID, name, and enrollment status.

The `else` statement on **Line 136** closes the `if` statement back on **Line 36**. As a reminder, this `if` statement checks to see if the student has already been enrolled — the `else` statement therefore catches if the student is *already* in the database and trying to enroll again. If that case happens we simply inform the user that they have already been enrolled and skip any face detection and localization.

6.4.2 Enrolling Faces

To enroll faces in our database, open up a terminal and execute the following command:

```
$ python enroll.py --id S1902 --name David --conf config/config.json
[INFO] warming up camera...
...
[INFO] 10 face images stored
[INFO] cleaning up
```

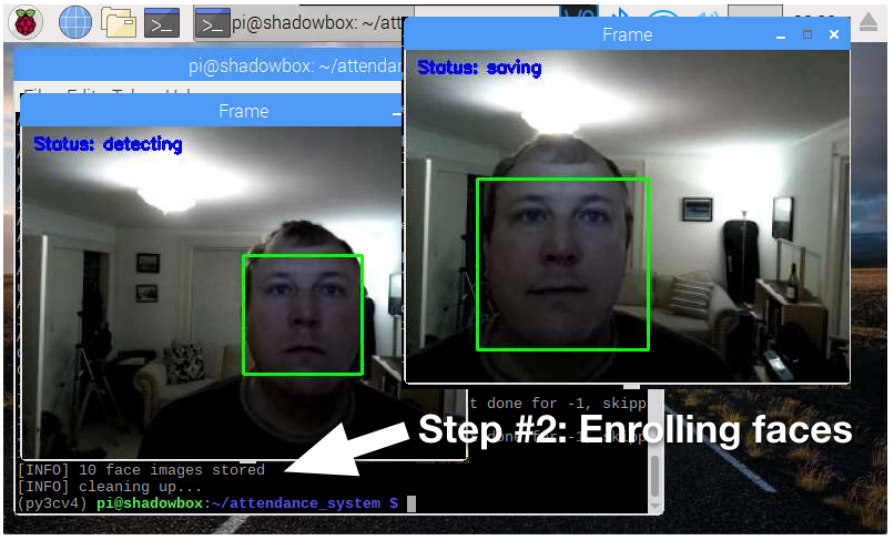

Figure 6.3: Step #2: Enrolling faces in our attendance system via `enroll.py`.

Figure 6.3 *(left)* shows the "face detection" status. During this phase our enrollment software is running face detection on each and every frame. Once we have reached a sufficient number of consecutive frames detected with a face, we change the status to "saving" (*right*) and begin

saving face images to disk. After we reach the required number of face images an audio message is played over the speaker and a notification is printed in the terminal.

Upon executing the script you can check the `dataset/pyimagesearch_gurus/S1902/` directory and see that there are ten face examples:

```
$ ls dataset/pyimagesearch_gurus/S1902
00000.png   00002.png   00004.png   00006.png   00008.png
00001.png   00003.png   00005.png   00007.png   00009.png
```

You can repeat the process of face enrollment via `enroll.py` for *each* student that is registered to the class.

Once you have face images for each student we can then train a face recognition model in Section 6.5.

6.4.3 Implementing Face Un-enrollment

If a student decides to drop out of the class we need to un-enroll them from both our (1) database and (2) face recognition model. To accomplish both these tasks we'll be using the `unenroll.py` script — open up that file now and insert the following code:

```
1  # import the necessary packages
2  from pyimagesearch.utils import Conf
3  from tinydb import TinyDB
4  from tinydb import where
5  import argparse
6  import shutil
7  import os
8
9  # construct the argument parser and parse the arguments
10 ap = argparse.ArgumentParser()
11 ap.add_argument("-i", "--id", required=True,
12     help="Unique student ID of the student")
13 ap.add_argument("-c", "--conf", required=True,
14     help="Path to the input configuration file")
15 args = vars(ap.parse_args())
```

Lines 2-7 import our required Python packages while **Lines 10-15** parse our command line arguments. We need two command line arguments here, `--id`, the ID of the student we are un-enrolling and `---conf`, the path to our configuration file.

We can then load the `Conf` and then access the students table:

```
17 # load the configuration file
18 conf = Conf(args["conf"])
19
20 # initialize the database and student table objects
21 db = TinyDB(conf["db_path"])
22 studentTable = db.table("student")
23
24 # retrieve the student document from the database, mark the student
25 # as unenrolled, and write back the document to the database
26 student = studentTable.search(where(args["id"]))
27 student[0][args["id"]][1] = "unenrolled"
28 studentTable.write_back(student)
29
30 # delete the student's data from the dataset
31 shutil.rmtree(os.path.join(conf["dataset_path"], conf["class"],
32     args["id"]))
33 print("[INFO] Please extract the embeddings and re-train the face" \
34     " recognition model...")
35
36 # close the database
37 db.close()
```

Line 26 searches the `studentTable` using the supplied `--id`.

Once we find the row we update the enrollment status to be *"unenrolled"*. We then delete the students face images from our `dataset` directory (**Lines 31 and 32**). The `db` is then serialized back out to disk on **Line 37**.

6.4.4 Un-enrolling Faces

Let's go ahead and un-enroll the "Adrian" student that we enrolled from Section 6.4.2:

```
$ python unenroll.py --id S1901 --conf config/config.json
[INFO] Please extract the embeddings and re-train the face recognition
model...
```

Checking the contents of `dataset/pyimagesearch_gurus/` you will no longer find a `S1901` directory — this is because the `S1901` student has been removed from our database.

You can use this script whenever you need to un-enroll a student from a database, but before you continue on to the next section, make sure you use the `enroll.py` script to register *at least two* students in the database. Once you have done so you can move on to training the actual face recognition component of the smart attendance system.

6.5 Step #3: Training the Face Recognition Component

Now that we have example images for each student in the class we can move on to training the face recognition component of the project.

6.5.1 Implementing Face Embedding Extraction

The `encode_faces.py` script we'll be reviewing to section is essentially identical to the script covered in Chapter 5. We'll still review the file here as a matter of completeness, but make sure you refer to Chapter 5 for more details on how this script works.

Open up the `encode_faces.py` file and insert the following code:

```
1  # import the necessary packages
2  from pyimagesearch.utils import Conf
3  from imutils import paths
4  import face_recognition
5  import argparse
6  import pickle
7  import cv2
8  import os
9
10 # construct the argument parser and parse the arguments
11 ap = argparse.ArgumentParser()
12 ap.add_argument("-c", "--conf", required=True,
13     help="Path to the input configuration file")
14 args = vars(ap.parse_args())
15
16 # load the configuration file
17 conf = Conf(args["conf"])
18
19 # grab the paths to the input images in our dataset
20 print("[INFO] quantifying faces...")
21 imagePaths = list(paths.list_images(
22     os.path.join(conf["dataset_path"], conf["class"])))
23
24 # initialize the list of known encodings and known names
25 knownEncodings = []
26 knownNames = []
```

Lines 2-8 import our required Python packages. The `face_recognition` library, in-conjunction with `dlib`, will be used to quantify each of the faces in our `dataset/` directory.

We then parse the path to our `--conf` file on (**Lines 11-14**). The configuration itself is loaded on **Line 17**.

Lines 21-22 grabs the paths to all images inside the `dataset/` directory. We then initialize

two lists, one to store the *quantifications* of each face followed by a second list to store the actual *names* of each face.

Let's loop over each of the `imagePaths`:

```
28 # loop over the image paths
29 for (i, imagePath) in enumerate(imagePaths):
30     # extract the person name from the image path
31     print("[INFO] processing image {}/{}".format(i + 1,
32         len(imagePaths)))
33     name = imagePath.split(os.path.sep)[-2]
34
35     # load the input image and convert it from RGB (OpenCV ordering)
36     # to dlib ordering (RGB)
37     image = cv2.imread(imagePath)
38     rgb = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
39
40     # compute the facial embedding for the face
41     encodings = face_recognition.face_encodings(rgb)
42
43     # loop over the encodings
44     for encoding in encodings:
45         # add each encoding + name to our set of known names and
46         # encodings
47         knownEncodings.append(encoding)
48         knownNames.append(name)
```

Line 33 extracts the `name` of the student from the `imagePath`. In this case the `name` is actually the ID of the student.

Line 37 and 38 reads our input image from disk and converts it from BGR to RGB channel ordering (the channel ordering that the `face_recognition` library expects when performing face quantification).

A call to the `face_encodings` method uses a neural network to compute a list of 128 floating point values used to quantify the face in the image. We then update our `knownEncodings` with each `encoding` and the `knownNames` list with the `name` of the person.

Finally, we can serialize both the `knownEncodings` and `knownNames` to disk:

```
50 # dump the facial encodings + names to disk
51 print("[INFO] serializing encodings...")
52 data = {"encodings": knownEncodings, "names": knownNames}
53 f = open(conf["encodings_path"], "wb")
54 f.write(pickle.dumps(data))
55 f.close()
```

Again, for more details on how the face encoding process works, refer to Chapter 5.

6.5.2 Extracting Face Embedding

To quantify each student face in the `dataset/` directory, open up a terminal and execute the following command:

```
$ python encode_faces.py --conf config/config.json
[INFO] quantifying faces...
[INFO] processing image 1/50
[INFO] processing image 2/50
[INFO] processing image 3/50
...
[INFO] processing image 48/50
[INFO] processing image 49/50
[INFO] processing image 50/50
[INFO] serializing encodings...
```

It is recommended to execute this script on your *laptop*, *desktop*, or GPU-enabled machine. It is possible to run the script on a RPi 3B+, 4B, or higher, although not recommended. Again, use a separate system to compute these embeddings and train the face recognition model in Section 6.5 — you can then take the resulting model file and transfer it back to the RPi.

6.5.3 Implementing the Training Script

Similar to our `encode_faces.py`, the `train_model.py` file is essentially identical to the script in Chapter 5. I'll review `train_model.py` here as well, but again, you should refer to Chapter **??** if you require more detail on what this script is doing.

Let's review `train_model.py` now:

```
1  # import the necessary packages
2  from pyimagesearch.utils import Conf
3  from sklearn.preprocessing import LabelEncoder
4  from sklearn.svm import SVC
5  import argparse
6  import pickle
7
8  # construct the argument parser and parse the arguments
9  ap = argparse.ArgumentParser()
10 ap.add_argument("-c", "--conf", required=True,
11     help="Path to the input configuration file")
12 args = vars(ap.parse_args())
13
14 # load the configuration file
15 conf = Conf(args["conf"])
16
17 # load the face encodings
```

```
18 print("[INFO] loading face encodings...")
19 data = pickle.loads(open(conf["encodings_path"], "rb").read())
20
21 # encode the labels
22 print("[INFO] encoding labels...")
23 le = LabelEncoder()
24 labels = le.fit_transform(data["names"])
```

Lines 9-12 parse the `--conf` switch. We then load the associated `Conf` file on **Line 15**.

Line 19 loads the serialized `data` from disk. The `data` includes both the (1) 128-d quantifications for each face, and (2) the names of each respective individual. We take the names then then pass them through a `LabelEncoder`, ensuring that each name (string) is represented by a unique integer.

We then train a SVM on the associated `data` and `labels`:

```
26 # train the model used to accept the 128-d encodings of the face and
27 # then produce the actual face recognition
28 print("[INFO] training model...")
29 recognizer = SVC(C=1.0, kernel="linear", probability=True)
30 recognizer.fit(data["encodings"], labels)
31
32 # write the actual face recognition model to disk
33 print("[INFO] writing the model to disk...")
34 f = open(conf["recognizer_path"], "wb")
35 f.write(pickle.dumps(recognizer))
36 f.close()
37
38 # write the label encoder to disk
39 f = open(conf["le_path"], "wb")
40 f.write(pickle.dumps(le))
41 f.close()
```

After training is complete we serialize both the face `recognizer` model and the `LabelEncoder` to disk.

Again, for more details on this script and how it works, make sure you refer to Chapter 5.

6.5.4 Running the Training Script

To train our face recognition model, execute the following command:

```
$ python train_model.py --conf config/config.json
[INFO] loading face encodings...
[INFO] encoding labels...
```

```
[INFO] training model...
[INFO] writing the model to disk...
```

Training should only take a few minutes. After training is complete you should have two new files in your `output/` directory, `recognizer.pickle` and `le.pickle` in addition to `encodings.pickle` from previously:

```
$ ls output
encodings.pickle    le.pickle       recognizer.pickle
```

The `recognizer.py` file is your actual trained SVM. The SVM model will be used to accept the 128-d face encoding inputs and then *predict* the probability of the student based on the face quantification.

We then take the prediction with the highest probability and pass it through our serialized `LabelEncoder` (i.e., `le.pickle`) to convert the prediction to a human-readable name (i.e., the unique ID of the student).

6.6 Step #4: Implementing the Attendance Script

We now have all the pieces of the puzzle — it's time to assemble them and create our smart attendance system.

Open up the `attendance.py` file and insert the following code:

```
# import the necessary packages
from pyimagesearch.utils import Conf
from imutils.video import VideoStream
from datetime import datetime
from datetime import date
from tinydb import TinyDB
from tinydb import where
import face_recognition
import numpy as np
import argparse
import imutils
import pyttsx3
import pickle
import time
import cv2

# construct the argument parser and parse the arguments
ap = argparse.ArgumentParser()
ap.add_argument("-c", "--conf", required=True,
```

```
20      help="Path to the input configuration file")
21  args = vars(ap.parse_args())
```

On **Lines 2-15** we import our required packages. Notable imports include `tinydb` used to interface with our `attendance.json` database, `face_recognition` to facilitate both face detection and face identification, and `pyttsx3` used for Text-to-Speech.

Lines 18-21 parse our command line arguments.

We can now move on to loading the configuration and accessing individual tables via TinyDB:

```
23  # load the configuration file
24  conf = Conf(args["conf"])
25
26  # initialize the database, student table, and attendance table
27  # objects
28  db = TinyDB(conf["db_path"])
29  studentTable = db.table("student")
30  attendanceTable = db.table("attendance")
31
32  # load the actual face recognition model along with the label encoder
33  recognizer = pickle.loads(open(conf["recognizer_path"], "rb").read())
34  le = pickle.loads(open(conf["le_path"], "rb").read())
```

Lines 29 and 30 grab a reference to the `student` and `attednance` tables, respectively. We then load the trained face `recognizer` model and `LabelEncoder` on **Lines 33 and 34**.

Let's access our video stream and perform a few more initializations:

```
36  # initialize the video stream and allow the camera sensor to warmup
37  print("[INFO] warming up camera...")
38  # vs = VideoStream(src=0).start()
39  vs = VideoStream(usePiCamera=True).start()
40  time.sleep(2.0)
41
42  # initialize previous and current person to None
43  prevPerson = None
44  curPerson = None
45
46  # initialize consecutive recognition count to 0
47  consecCount = 0
48
49  # initialize the text-to-speech engine, set the speech language, and
50  # the speech rate
51  print("[INFO] taking attendance...")
52  ttsEngine = pyttsx3.init()
53  ttsEngine.setProperty("voice", conf["language"])
```

```
54 ttsEngine.setProperty("rate", conf["rate"])
55
56 # initialize a dictionary to store the student ID and the time at
57 # which their attendance was taken
58 studentDict = {}
```

Line 43 and 44 initialize two variables, `prevPerson`, the ID of the *previous* person recognized in the video stream, and `curPerson`, the ID of the *current* person identified in the stream. In order to reduce false-positive identifications we'll ensure that the `prevPerson` and `curPerson` match for a total of `consec_count` frames (defined inside our `config.json` file from Section 6.2.3).

The `consecCount` integer keeps track of the number of consecutive frames with the *same* person identified.

Lines 52-54 initialize our `ttsEngine`, used to generate speech and play it through our speakers.

We then initialize `studentDict`, a dictionary used to map a student ID to when their respective attendance was taken.

We're now entering the body of our script:

```
60 # loop over the frames from the video stream
61 while True:
62     # store the current time and calculate the time difference
63     # between the current time and the time for the class
64     currentTime = datetime.now()
65     timeDiff = (currentTime - datetime.strptime(conf["timing"],
66         "%H:%M")).seconds
67
68     # grab the next frame from the stream, resize it and flip it
69     # horizontally
70     frame = vs.read()
71     frame = imutils.resize(frame, width=400)
72     frame = cv2.flip(frame, 1)
```

Line 63 grabs the current time. We then take this value and compute the difference between the *current time* and *when class officially starts*. We'll use this `timeDiff` value to determine if class has already started, in which time to take attendance has closed.

Line 70 reads a `frame` from our video stream which we then preprocess by resizing to have a width of 400px and then flipping horizontally.

Let's check to see if the maximum time limit to take attendance has been reached:

```
74     # if the maximum time limit to record attendance has been crossed
```

```
75      # then skip the attendance taking procedure
76      if timeDiff > conf["max_time_limit"]:
77          # check if the student dictionary is not empty
78          if len(studentDict) != 0:
79              # insert the attendance into the database and reset the
80              # student dictionary
81              attendanceTable.insert({str(date.today()): studentDict})
82              studentDict = {}
83
84          # draw info such as class, class timing, and current time on
85          # the frame
86          cv2.putText(frame, "Class: {}".format(conf["class"]),
87              (10, 10), cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0, 0, 255), 1)
88          cv2.putText(frame, "Class timing: {}".format(conf["timing"]),
89              (10, 25), cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0, 0, 255), 1)
90          cv2.putText(frame, "Current time: {}".format(
91              currentTime.strftime("%H:%M:%S")), (10, 40),
92              cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0, 0, 255), 1)
93
94          # show the frame
95          cv2.imshow("Attendance System", frame)
96          key = cv2.waitKey(1) & 0xFF
97
98          # if the `q` key was pressed, break from the loop
99          if key == ord("q"):
100             break
101
102         # skip the remaining steps since the time to take the
103         # attendance has ended
104         continue
```

Provided that (1) class has *already* started, and (2) the maximum time limit fdor attendance has been passed (**Line 76**), we make a second check on **Line 78** to see if the attendance record has been added to our database. If we have *not* added the attendance results to our database, we insert a new set of records to the database, indicating that each of the individual students in `studentDict` are in attendance.

Lines 86-92 draw class information on our `frame`, including the name of the class, when class starts, and the current timestamp.

The remaining code blocks assume that we are *still taking attendance*, implying that we're not past the attendance time limit:

```
106     # convert the frame from RGB (OpenCV ordering) to dlib
107     # ordering (RGB)
108     rgb = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)
109
110     # detect the (x, y)-coordinates of the bounding boxes
111     # corresponding to each face in the input image
112     boxes = face_recognition.face_locations(rgb,
```

```
113             model=conf["detection_method"])
114
115         # loop over the face detections
116         for (top, right, bottom, left) in boxes:
117             # draw the face detections on the frame
118             cv2.rectangle(frame, (left, top), (right, bottom),
119                 (0, 255, 0), 2)
120
121         # calculate the time remaining for attendance to be taken
122         timeRemaining = conf["max_time_limit"] - timeDiff
123
124         # draw info such as class, class timing, current time, and
125         # remaining attendance time on the frame
126         cv2.putText(frame, "Class: {}".format(conf["class"]), (10, 10),
127             cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0, 0, 255), 1)
128         cv2.putText(frame, "Class timing: {}".format(conf["timing"]),
129             (10, 25), cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0, 0, 255), 1)
130         cv2.putText(frame, "Current time: {}".format(
131             currentTime.strftime("%H:%M:%S")), (10, 40),
132             cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0, 0, 255), 1)
133         cv2.putText(frame, "Time remaining: {}s".format(timeRemaining),
134             (10, 55), cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0, 0, 255), 1)
```

Line 108 converts our `frame` to RGB ordering so we can perform both face detection and recognition via `dlib` and `face_recognition`.

Lines 112 and 113 perform face detection. We then loop over each of the detecting bounding `boxes` and draw them on our `frame`.

Lines 126-134 draw class information on our screen, most importantly, the amount of time remaining to register yourself as having "attended" the class.

Let's now check and see if any faces were detected in the current frame:

```
136         # check if atleast one face has been detected
137         if len(boxes) > 0:
138             # compute the facial embedding for the face
139             encodings = face_recognition.face_encodings(rgb, boxes)
140
141             # perform classification to recognize the face
142             preds = recognizer.predict_proba(encodings)[0]
143             j = np.argmax(preds)
144             curPerson = le.classes_[j]
145
146             # if the person recognized is the same as in the previous
147             # frame then increment the consecutive count
148             if prevPerson == curPerson:
149                 consecCount += 1
150
151             # otherwise, these are two different people so reset the
152             # consecutive count
```

```
153         else:
154             consecCount = 0
155
156         # set current person to previous person for the next
157         # iteration
158         prevPerson = curPerson
```

Line 139 takes all detected faces and then extracts 128-d embeddings used to quantify each face. We take these face embeddings and pass them through our `recognizer`, finding the index of the label with the largest corresponding probability (**Lines 142-144**).

Line 148 checks to see if the `prevPerson` prediction matches the `curPerson` prediction, in which case we increment the `consecCount`. Otherwise, we do not have matching consecutive predictions so we reset the `consecCount` (**Lines 153 and 154**).

Once `consecCount` reaches a suitable threshold, indicating a positive identification, we can process the student:

```
160         # if a particular person is recognized for a given
161         # number of consecutive frames, we have reached a
162         # positive recognition and alert/greet the person accordingly
163         if consecCount >= conf["consec_count"]:
164             # check if the student's attendance has been already
165             # taken, if not, record the student's attendance
166             if curPerson not in studentDict.keys():
167                 studentDict[curPerson] = datetime.now().strftime("%H:%M:%S")
168
169                 # get the student's name from the database and let them
170                 # know that their attendance has been taken
171                 name = studentTable.search(where(
172                     curPerson))[0][curPerson][0]
173                 ttsEngine.say("{} your attendance has been taken.".format(
174                     name))
175                 ttsEngine.runAndWait()
176
177             # construct a label saying the student has their attendance
178             # taken and draw it on to the frame
179             label = "{}, you are now marked as present in {}".format(
180                 name, conf["class"])
181             cv2.putText(frame, label, (5, 175),
182                 cv2.FONT_HERSHEY_SIMPLEX, 0.5, (255, 0, 0), 2)
183
184         # otherwise, we have not reached a positive recognition and
185         # ask the student to stand in front of the camera
186         else:
187             # construct a label asking the student to stand in fron
188             # to the camera and draw it on to the frame
189             label = "Please stand in front of the camera"
190             cv2.putText(frame, label, (5, 175),
191                 cv2.FONT_HERSHEY_SIMPLEX, 0.5, (255, 0, 0), 2)
```

Line 163 ensures that the consecutive prediction count has been satisfied (used to reduce false-positive identifications). We then check to see if the student's attendance has already been taken (**Line 166**), and if *not*, we update the `studentDict` to include (1) the ID of the person, and (2)the timestamp at which attendance was taken.

Lines 171-175 lookup the `name` of the student via the ID and then use the TTS engine to let the student know their attendance has been taken.

Line 186 handles when the `consecCount` threshold has *not* been met — in that case we tell the user to stand in front of the camera until their attendance has been taken.

Our final code block handles displaying the frame to our screen and a few tidying up operations:

```
193     # show the frame
194     cv2.imshow("Attendance System", frame)
195     key = cv2.waitKey(1) & 0xFF
196
197     # check if the `q` key was pressed
198     if key == ord("q"):
199         # check if the student dictionary is not empty, and if so,
200         # insert the attendance into the database
201         if len(studentDict) != 0:
202             attendanceTable.insert({str(date.today()): studentDict})
203
204         # break from the loop
205         break
206
207 # clean up
208 print("[INFO] cleaning up...")
209 vs.stop()
210 db.close()
```

In the event the `q` key is pressed, we check to see if there are any students in `studentDict` that need to have their attendance recorded, and if so, we insert them into our TinyDB — after which we `break` from the loop(**Lines 198-205**).

6.7 Smart Attendance System Results

It's been quite a journey to get here, but we are now ready to run our smart attendance system on the Raspberry Pi!

Open up a terminal and execute the following command:

```
$ python attendance.py --conf config/config.json
[INFO] warming up camera...
```

```
[INFO] taking attendance...
[INFO] cleaning up...
```

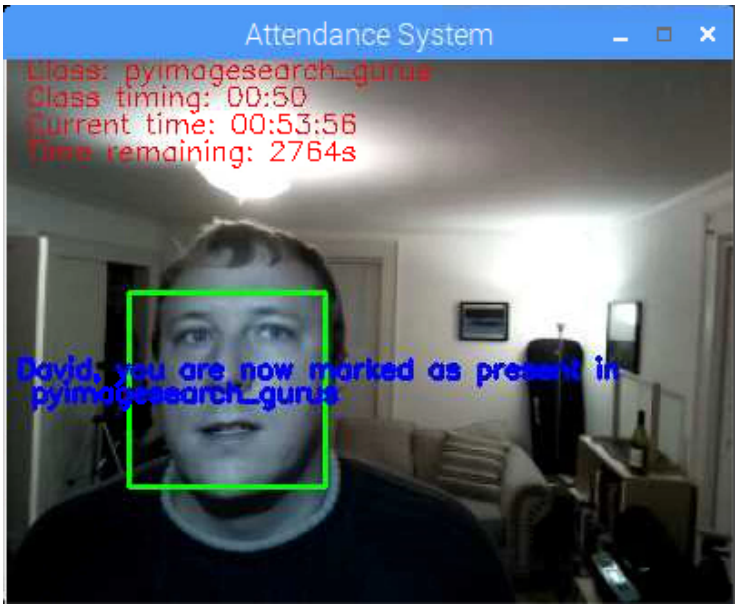

Figure 6.4: An example of a student enrolled in the PyImageSearch Gurus course being marked as "present" in the TinyDB database. Face detection and face recognition has recognized this student while sounding an audible message and printing a text based annotation on the screen.

Figure 6.4 demonstrates our smart attendance system in action. As students enter the classroom, attendance is taken until the time expires. Each result is saved to our TinyDB database. The instructor can can query the database at a later date to determine which students have attended/not attended certain class sessions throughout the semester.

6.8 Summary

In this chapter you learned how to implement a smart attendance application from scratch.

This system is capable of running in real-time on the Raspberry Pi, *despite* using more advanced computer vision and deep learning techniques.

I know this has been a heavily requested topic on the PyImageSearch blog, particularly among students working on their final projects before graduation, so if you use any of the concepts/code in this chapter, please don't forget to cite it in your final reports. You can find citation/reference instructions here on the PyImageSearch blog: http://pyimg.co/hwovx.

It looks like you have reached the end of the Table of Contents and Sample Chapters!

To order your copy of *Raspberry Pi for Computer Vision*, **just click the button below.**

After you order your copy, you will:

1. Be able to *immediately* download all chapters, code examples, and datasets associated with the text.
2. Download my pre-configured Raspbian .img file with all the necessary CV and DL libraries you need pre-installed. Just flash the .img file to your micro-SD card and boot!
3. Have access to myself and the rest of the PyImageSearch team to ask questions.

And did I mention that you'll receive *free* digital release updates *for life* as well?

Just click the button below to order your copy.

Click Here to Order Your