

# OCR with OpenCV, Tesseract, and Python

OCR Practitioner Bundle, 1st Edition (version 1.0)

Adrian Rosebrock, PhD

Abhishek Thanki

Sayak Paul

Jon Haase



The contents of this book, unless otherwise indicated, are Copyright ©2020 Adrian Rosebrock, [PyImageSearch.com](https://www.pyimagesearch.com). All rights reserved. Books like this are made possible by the time invested by the authors. If you received this book and did not purchase it, please consider making future books possible by buying a copy at <https://www.pyimagesearch.com/books-and-courses/> today.

# Contents

<b>Contents</b>	iii
<b>Companion Website</b>	xiii
<b>Acronyms</b>	xv
<b>1 Introduction</b>	1
1.1 Book Organization . . . . .	2
1.2 What You Will Learn . . . . .	2
1.3 Summary . . . . .	3
<b>2 Training an OCR Model with Keras and TensorFlow</b>	5
2.1 Chapter Learning Objectives . . . . .	5
2.2 OCR with Keras and TensorFlow . . . . .	6
2.2.1 Our Digits and Letters Dataset . . . . .	6
2.2.2 Project Structure . . . . .	7
2.2.3 Implementing Our Dataset Loader Functions . . . . .	8
2.2.4 Implementing Our Training Script . . . . .	11
2.2.5 Training Our Keras and TensorFlow OCR Model . . . . .	19
2.3 Summary . . . . .	21
<b>3 Handwriting Recognition with Keras and TensorFlow</b>	23
3.1 Chapter Learning Objectives . . . . .	24
3.2 OCR'ing Handwriting with Keras and TensorFlow . . . . .	24
3.2.1 What Is Handwriting Recognition? . . . . .	25
3.2.2 Project Structure . . . . .	26
3.2.3 Implementing Our Handwriting Recognition Script . . . . .	26
3.2.4 Handwriting Recognition Results . . . . .	32
3.2.5 Limitations, Drawbacks, and Next Steps . . . . .	35

3.3	Summary . . . . .	36
<b>4</b>	<b>Using Machine Learning to Denoise Images for Better OCR Accuracy</b>	<b>37</b>
4.1	Chapter Learning Objectives . . . . .	38
4.2	Image Denoising with Machine Learning . . . . .	38
4.2.1	Our Noisy Document Dataset . . . . .	39
4.2.2	The Denoising Document Algorithm . . . . .	40
4.2.3	Project Structure . . . . .	42
4.2.4	Implementing Our Configuration File . . . . .	44
4.2.5	Creating Our Blur and Threshold Helper Function . . . . .	45
4.2.6	Implementing the Feature Extraction Script . . . . .	47
4.2.7	Running the Feature Extraction Script . . . . .	51
4.2.8	Implementing Our Denoising Training Script . . . . .	52
4.2.9	Training Our Document Denoising Model . . . . .	54
4.2.10	Creating the Document Denoiser Script . . . . .	54
4.2.11	Running our Document Denoiser . . . . .	57
4.3	Summary . . . . .	59
<b>5</b>	<b>Making OCR "Easy" with EasyOCR</b>	<b>61</b>
5.1	Chapter Learning Objectives . . . . .	62
5.2	Getting Started with EasyOCR . . . . .	62
5.2.1	What Is EasyOCR? . . . . .	62
5.2.2	Installing EasyOCR . . . . .	63
5.3	Using EasyOCR for Optical Character Recognition . . . . .	64
5.3.1	Project Structure . . . . .	64
5.3.2	Implementing our EasyOCR Script . . . . .	65
5.3.3	EasyOCR Results . . . . .	67
5.4	Summary . . . . .	70
<b>6</b>	<b>Image/Document Alignment and Registration</b>	<b>71</b>
6.1	Chapter Learning Objectives . . . . .	72
6.2	Document Alignment and Registration with OpenCV . . . . .	72
6.2.1	What Is Document Alignment and Registration? . . . . .	72
6.2.2	How Can OpenCV Be Used for Document Alignment? . . . . .	74
6.2.3	Project Structure . . . . .	76
6.2.4	Aligning Documents with Keypoint Matching . . . . .	77
6.2.5	Implementing Our Document Alignment Script . . . . .	81

6.2.6	Image and Document Alignment Results . . . . .	83
6.3	Summary . . . . .	86
<b>7</b>	<b>OCR'ing a Document, Form, or Invoice</b>	<b>87</b>
7.1	Automatically Registering and OCR'ing a Form . . . . .	88
7.1.1	Why Use OCR on Forms, Invoices, and Documents? . . . . .	88
7.1.2	Steps to Implementing a Document OCR Pipeline with OpenCV and Tesseract . . . . .	88
7.1.3	Project Structure . . . . .	91
7.1.4	Implementing Our Document OCR Script with OpenCV and Tesseract . . . . .	91
7.1.5	OCR Results Using OpenCV and Tesseract . . . . .	99
7.2	Summary . . . . .	103
<b>8</b>	<b>Sudoku Solver and OCR</b>	<b>105</b>
8.1	Chapter Learning Objectives . . . . .	106
8.2	OpenCV Sudoku Solver and OCR . . . . .	106
8.2.1	How to Solve Sudoku Puzzles with OpenCV and OCR . . . . .	107
8.2.2	Configuring Your Development Environment to Solve Sudoku Puzzles with OpenCV and OCR . . . . .	108
8.2.3	Project Structure . . . . .	108
8.2.4	SudokuNet: A Digit OCR Model Implemented in Keras and TensorFlow . . . . .	109
8.2.5	Implementing with Keras and TensorFlow . . . . .	111
8.2.6	Training Our Sudoku Digit Recognizer with Keras and TensorFlow . . . . .	114
8.2.7	Finding the Sudoku Puzzle Board in an Image with OpenCV . . . . .	115
8.2.8	Extracting Digits from a Sudoku Puzzle with OpenCV . . . . .	121
8.2.9	Implementing Our OpenCV Sudoku Puzzle Solver . . . . .	123
8.2.10	OpenCV Sudoku Puzzle Solver OCR Results . . . . .	128
8.2.11	Credits . . . . .	130
8.3	Summary . . . . .	130
<b>9</b>	<b>Automatically OCR'ing Receipts and Scans</b>	<b>131</b>
9.1	Chapter Learning Objectives . . . . .	132
9.2	OCR'ing Receipts with OpenCV and Tesseract . . . . .	132
9.2.1	Project Structure . . . . .	132
9.2.2	Implementing Our Receipt Scanner . . . . .	132
9.2.3	Receipt Scanner and OCR Results . . . . .	140
9.3	Summary . . . . .	142

<b>10 OCR'ing Business Cards</b>	<b>143</b>
10.1 Chapter Learning Objectives . . . . .	143
10.2 Business Card OCR . . . . .	144
10.2.1 Project Structure . . . . .	144
10.2.2 Implementing Business Card OCR . . . . .	144
10.2.3 Business Card OCR Results . . . . .	149
10.3 Summary . . . . .	152
<b>11 OCR'ing License Plates with ANPR/ALPR</b>	<b>153</b>
11.1 Chapter Learning Objectives . . . . .	154
11.2 ALPR/ANPR and OCR . . . . .	154
11.2.1 What Is Automatic License Plate Recognition? . . . . .	155
11.2.2 Project Structure . . . . .	156
11.2.3 Implementing ALPR/ANPR with OpenCV and OCR . . . . .	157
11.2.4 Debugging Our Computer Vision Pipeline . . . . .	158
11.2.5 Locating Potential License Plate Candidates . . . . .	159
11.2.6 Pruning License Plate Candidates . . . . .	164
11.2.7 Defining Our Tesseract ANPR Options . . . . .	167
11.2.8 The Heart of the ANPR Implementation . . . . .	168
11.2.9 Creating Our ANPR and OCR Driver Script . . . . .	169
11.2.10 Automatic License Plate Recognition Results . . . . .	172
11.2.11 Limitations and Drawbacks . . . . .	174
11.3 Summary . . . . .	174
<b>12 Multi-Column Table OCR</b>	<b>177</b>
12.1 Chapter Learning Objectives . . . . .	178
12.2 OCR'ing Multi-Column Data . . . . .	178
12.2.1 Our Multi-Column OCR Algorithm . . . . .	178
12.2.2 Project Structure . . . . .	180
12.2.3 Installing Required Packages . . . . .	181
12.2.4 Implementing Multi-Column OCR . . . . .	182
12.2.5 Multi-Column OCR Results . . . . .	192
12.3 Summary . . . . .	195
<b>13 Blur Detection in Text and Documents</b>	<b>197</b>
13.1 Chapter Learning Objectives . . . . .	197
13.2 Detecting Blurry Text and Documents . . . . .	198

13.2.1 What Is Blur Detection and Why Do We Want to Detect Blur? . . . . .	198
13.2.2 What Is the Fast Fourier Transform? . . . . .	200
13.3 Implementing Our Text Blur Detector . . . . .	201
13.3.1 Project Structure . . . . .	201
13.3.2 Creating the Blur Detector Helper Function . . . . .	202
13.3.3 Detecting Text and Document Blur . . . . .	204
13.3.4 Blurry Text and Document Detection Results . . . . .	206
13.4 Summary . . . . .	207
<b>14 OCR'ing Video Streams</b>	<b>209</b>
14.1 Chapter Learning Objectives . . . . .	209
14.2 OCR'ing Real-Time Video Streams . . . . .	210
14.2.1 Project Structure . . . . .	210
14.2.2 Implementing Our Video Writer Utility . . . . .	211
14.2.3 Implementing Our Real-Time Video OCR Script . . . . .	214
14.2.4 Real-Time Video OCR Results . . . . .	222
14.3 Summary . . . . .	223
<b>15 Improving Text Detection Speed with OpenCV and GPUs</b>	<b>225</b>
15.1 Chapter Learning Objectives . . . . .	225
15.2 Using Your GPU for OCR with OpenCV . . . . .	226
15.2.1 Project Structure . . . . .	226
15.2.2 Implementing Our OCR GPU Benchmark Script . . . . .	226
15.2.3 Speed Test: OCR With and Without GPU . . . . .	229
15.2.4 OCR on GPU for Real-Time Video Streams . . . . .	230
15.2.5 GPU and OCR Results . . . . .	235
15.3 Summary . . . . .	235
<b>16 Text Detection and OCR with Amazon Rekognition API</b>	<b>237</b>
16.1 Chapter Learning Objectives . . . . .	238
16.2 Amazon Rekognition API for OCR . . . . .	238
16.2.1 Obtaining Your AWS Rekognition Keys . . . . .	239
16.2.2 Installing Amazon's Python Package . . . . .	239
16.2.3 Project Structure . . . . .	239
16.2.4 Creating Our Configuration File . . . . .	240
16.2.5 Implementing the Amazon Rekognition OCR Script . . . . .	240
16.2.6 Amazon Rekognition OCR Results . . . . .	244

16.3 Summary . . . . .	246
<b>17 Text Detection and OCR with Microsoft Cognitive Services</b>	<b>249</b>
17.1 Chapter Learning Objectives . . . . .	249
17.2 Microsoft Cognitive Services for OCR . . . . .	250
17.2.1 Obtaining Your Microsoft Cognitive Services Keys . . . . .	250
17.2.2 Project Structure . . . . .	250
17.2.3 Creating Our Configuration File . . . . .	251
17.2.4 Implementing the Microsoft Cognitive Services OCR Script . . . . .	251
17.2.5 Microsoft Cognitive Services OCR Results . . . . .	255
17.3 Summary . . . . .	259
<b>18 Text Detection and OCR with Google Cloud Vision API</b>	<b>261</b>
18.1 Chapter Learning Objectives . . . . .	261
18.2 Google Cloud Vision API for OCR . . . . .	262
18.2.1 Obtaining Your Google Cloud Vision API Keys . . . . .	262
18.2.1.1 Prerequisite . . . . .	262
18.2.1.2 Steps to Enable Google Cloud Vision API and Download Cred- entials . . . . .	262
18.2.2 Configuring Your Development Environment for the Google Cloud Vi- sion API . . . . .	262
18.2.3 Project Structure . . . . .	263
18.2.4 Implementing the Google Cloud Vision API Script . . . . .	263
18.2.5 Google Cloud Vision API OCR Results . . . . .	267
18.3 Summary . . . . .	269
<b>19 Training a Custom Tesseract Model</b>	<b>271</b>
19.1 Chapter Learning Objectives . . . . .	272
19.2 Your Tesseract Training Development Environment . . . . .	272
19.2.1 Step #1: Installing Required Packages . . . . .	273
19.2.1.1 Ubuntu . . . . .	273
19.2.1.2 macOS . . . . .	273
19.2.2 Step #2: Building and Installing Tesseract Training Tools . . . . .	274
19.2.2.1 Ubuntu . . . . .	274
19.2.2.2 macOS . . . . .	275
19.2.3 Step #3: Cloning <code>tesstrain</code> and Installing Requirements . . . . .	276
19.3 Training Your Custom Tesseract Model . . . . .	277

19.3.1	Project Structure . . . . .	277
19.3.2	Our Tesseract Training Dataset . . . . .	278
19.3.3	Creating Your Tesseract Training Dataset . . . . .	279
19.3.4	Step #1: Setup the Base Model . . . . .	280
19.3.5	Step #2: Set Your TESSDATA_PREFIX . . . . .	281
19.3.6	Step #3: Setup Your Training Data . . . . .	282
19.3.7	Step #4: Fine-Tune the Tesseract Model . . . . .	283
19.3.8	Training Tips and Suggestions . . . . .	286
19.4	Summary . . . . .	287
<b>20</b>	<b>OCR'ing Text with Your Custom Tesseract Model</b>	<b>289</b>
20.1	Chapter Learning Objectives . . . . .	289
20.2	OCR with Your Custom Tesseract Model . . . . .	289
20.2.1	Project Structure . . . . .	290
20.2.2	Implementing Your Tesseract OCR Script . . . . .	290
20.2.3	Custom Tesseract OCR Results . . . . .	292
20.3	Summary . . . . .	296
<b>21</b>	<b>Conclusions</b>	<b>297</b>
21.1	Where to Now? . . . . .	299
21.2	Thank You . . . . .	300
<b>Bibliography</b>		<b>301</b>



*To my first computer science teacher, Marla Wood.*

*Thank you for everything you've done for me when I was a high school kid.*

*Rest assured, wherever you are, you're a big part*

*and a big reason for the person I am today.*



# Companion Website

Thank you for picking up a copy of *OCR with OpenCV, Tesseract, and Python!* To accompany this book, I have created a companion website which includes:

- **Up-to-date installation instructions** on how to configure your environment for OCR development
- Instructions on how to use the **pre-configured Oracle VirtualBox Virtual Machine** on your system
- **Supplementary materials** that we could not fit inside this book, including how to use pre-configured environments
- **Frequently Asked Questions (FAQs)** and their suggested fixes and solutions
- **Access to the PyImageSearch Community Forums** for discussion about OCR and other computer vision and deep learning topics (*Expert Bundle* only)

You can find the companion website here: <http://pyimg.co/ocrcw>



# Acronyms

**ALPR:** automatic license plate recognition

**Amazon S3:** Amazon Simple Storage Service

**ANPR:** automatic number plate recognition

**API:** application programming interface

**ASCII:** American Standard Code for Information Interchange

**ATR:** automatic target recognition

**AWS:** Amazon Web Services

**BGR:** Blue Green Red

**cd:** change directory

**CPU:** central processing unit

**CRNN:** convolutional recurrent neural network

**CSV:** comma-separated values

**CTC:** connectionist temporal classification

**d:** dimensional

**DFT:** discrete Fourier transform

**DL4CV:** *Deep Learning for Computer Vision with Python*

**dnn:** deep neural network

**EAST:** efficient and accurate scene text detector

**FFT:** fast Fourier transform

**FPS:** frames per second

**GIF:** graphics interchange format

**GCP:** Google Cloud Platform

**GFTT:** good features to track

**GIMP:** GNU image manipulation program

**GPU:** graphics processing unit

**gt:** ground-truth

**GUI:** graphical user interface

**HAC:** hierarchical agglomerative clustering

**HDF5:** Hierarchical Data Format version 5

**HOG:** histogram of oriented gradient

**HP:** Hewlett-Packard

**I/O:** input/output

**IDE:** integrated development environment

**JSON:** JavaScript Object Notation

**LSTM:** long short-term memory

**maxAR:** maximum aspect ratio

**MCS:** Microsoft Cognitive Services

**minAR:** minimum aspect ratio

**mph:** miles per hour

**NLP:** natural language processing

**NMS:** non-maximum suppression

**OCR:** optical character recognition

**OpenCV:** Open Source Computer Vision Library

**ORB:** oriented FAST and rotated BRIEF

**PIL:** Python Imaging Library

**PNG:** portable network graphics

**PSM:** page segmentation method

**PSM:** page segmentation mode

**R-CNN:** regions-convolutional neural network

**RANSAC:** random sample consensus

**REST:** representational state transfer

**RFR:** random forest regressor

**RGB:** Red Green Blue

**RMSE:** root-mean-square error

**RNN:** recurrent neural network

**ROI:** region of interest

**SDK:** software development kit

**segROI:** segment ROI

**SGD:** stochastic gradient descent

**SIFT:** scale-invariant feature transform

**sym-link:** symbolic link

**SPECT:** single-photon emission computerized tomography

**SSD:** single-shot detectors

**SVM:** support vector machine

**UMBC:** University of Maryland, Baltimore County

**URL:** Uniform Resource Locator

**VM:** virtual machine

**YOLO:** you only look once



# Chapter 1

## Introduction

Welcome to the “*OCR Practitioner Bundle*” of *OCR with OpenCV, Tesseract, and Python!* Take a second to congratulate yourself on completing the “*OCR Starter*” *Bundle* — you now have a strong foundation to learn more advanced optical character recognition (OCR).

While the “*OCR with OpenCV, Tesseract, and Python: Intro to OCR*” book focused primarily on image processing and computer vision techniques, with only a hint of deep learning, this volume flips the relationship — here, we’ll be focusing primarily on deep learning-based OCR techniques. Computer vision and image processing algorithms will be used, but mainly to facilitate deep learning.

Additionally, this volume starts to introduce solutions to real-world OCR problems, including:

- Automatic license/number plate recognition
- OCR’ing and scanning receipts
- Image/document alignment and registration
- Building software to OCR forms, invoices, and other documents.
- Automatically denoising images with machine learning (to improve OCR accuracy)

Just like the first OCR book, “*OCR with OpenCV, Tesseract, and Python: Intro to OCR*,” I place a **strong emphasis on learning by doing**. It’s not enough to follow this text — you need to open your terminal/code editor and execute the code to learn.

Merely reading a book or watching a tutorial is a form of passive learning. You only involve two of your senses: eyes and ears.

**Instead, what you need to master this material is active learning.** Use your eyes and ears as you follow along, but then take the next step to involve tactile sensations — use your keyboard to write code and run sample scripts.

Then use your highly intelligent mind (you wouldn't be reading this book if you weren't smart) to critically analyze the results. What would happen if you tweak this parameter over here? What if you adjusted the learning rate of your neural network? Or what happens if you apply the *same* technique to a *different* dataset?

The more senses and experiences you can involve in your learning, the easier it will be for you to master OCR.

## 1.1 Book Organization

The “*Intro to OCR*” *Bundle* was organized linearly, taking a stair-step approach to learning OCR. You would learn a new OCR technique, apply it to a set of sample images, and then learn how to extend that approach in the next chapter to improve accuracy and learn a new skill.

This volume is organized slightly differently. You'll still learn linearly; however, I've organized the chapters based on case studies that build on each other. Doing so allows you to jump to a set of chapters that you're *specifically* interested in — ideally, these chapters will help you complete your work project, perform novel research, or submit your final graduation project.

I still recommend that you read this book from start to finish (as that will ensure you maximize value), but at the same time, don't feel constrained to read it linearly. **This book is meant to be your playbook to solve real-world optical character recognition problems.**

## 1.2 What You Will Learn

As stated in the previous section, the “*OCR Practitioner*” *Bundle* takes a “project-based” approach to learn OCR. You will improve your OCR skills by working on actual real-world projects, including:

- Training a machine learning model to denoise documents, thereby improving OCR results
- Performing automatic image/document alignment and registration
- Building a project that uses image alignment and registration to OCR a *structured* document, such as a form, invoice, etc.
- Solving Sudoku puzzles with OCR
- Scanning and OCR'ing receipts
- Building an automatic license/number plate recognition (ALPR/ANPR) system

- OCR'ing real-time video streams
- Performing handwriting recognition
- Leveraging GPUs to improve text detection and OCR speed
- Training and fine-tuning Tesseract models on your custom datasets

By the end of this book, you will have the ability to approach real-world OCR projects confidently.

## 1.3 Summary

As you found out in the “*Intro to OCR*” *Bundle*, performing optical character recognition isn’t as easy as installing a library via `pip` and then letting the OCR engine do the magic with you. If it were that easy, this series of books wouldn’t exist!

Instead, you found out that OCR is more challenging. You need to understand the various modes and options inside of Tesseract. Furthermore, you need to know when and where to use each setting.

To improve OCR accuracy further, you sometimes need to leverage computer vision and image processing libraries such as OpenCV. The OpenCV library can help you pre-process your images, clean them up, and improve your OCR accuracy.

**I’ll end this chapter by asking you to keep in mind that this volume is meant to be your playbook for approaching OCR problems.** If you have a particular OCR project you’re trying to solve, you *can* jump immediately to that chapter; however, I *strongly encourage* you to read this book from start-to-finish still.

The field of OCR, just like computer vision as a whole, is like a toolbox. It’s not enough to fill your toolbox with the right tools — *you still need to know when and where to pull out your hammer, saw, screwdriver, etc.* Throughout the rest of this volume, you’ll gain practical experience and see firsthand when to use each tool.

Let’s get started.



## Chapter 2

# Training an OCR Model with Keras and TensorFlow

In this chapter, you will learn how to train an optical character recognition (OCR) model using Keras and TensorFlow. Then, in the next chapter, you'll learn how to take our trained model and then apply it to the task of handwriting recognition.

The goal of these two chapters is to obtain a deeper understanding of how deep learning is applied to the classification of handwriting, and more specifically, our goal is to:

- Become familiar with some well-known, readily available handwriting datasets for both digits and letters
- Understand how to train deep learning models to recognize handwritten digits and letters
- Gain experience in applying our custom-trained model to some real-world sample data
- Understand some of the challenges with noisy, real-world data and how we might want to augment our handwriting datasets to improve our model and results

We'll start with the fundamentals of using well-known handwriting datasets and training a ResNet deep learning model on these data.

Let's get started now!

### 2.1 Chapter Learning Objectives

In this chapter, you will:

- Review two separate character datasets, one for the characters  $0\text{--}9$  and then a second dataset for the characters  $A\text{--}Z$

- Combine both these datasets into a *single dataset* that we'll use for handwriting recognition
- Train ResNet to recognize each of these characters using Keras and TensorFlow

## 2.2 OCR with Keras and TensorFlow

In the first part of this chapter, we'll discuss the steps required to implement and train a custom OCR model with Keras and TensorFlow. We'll then examine the handwriting datasets that we'll use to train our model.

From there, we'll implement a couple of helper/utility functions that will aid us in loading our handwriting datasets from disk and then pre-processing them.

Given these helper functions, we'll be able to create our custom OCR training script with Keras and TensorFlow.

After training, we'll review the results of our OCR work.

### 2.2.1 Our Digits and Letters Dataset

To train our custom Keras and TensorFlow model, we'll be utilizing two datasets:

- i. The standard MNIST 0–9 dataset by LeCun et al. [1] (which is included in most popular Python deep learning libraries)
- ii. The Kaggle A–Z dataset [2] by Sachin Patel, based on the NIST Special Database 19 [3]

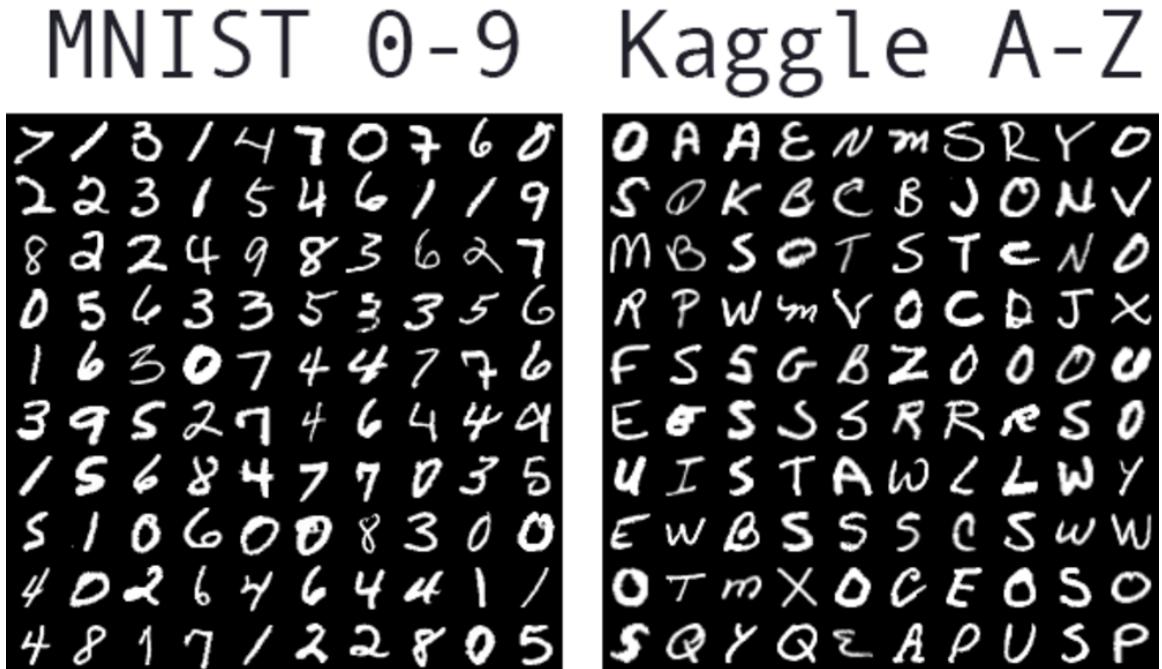
The **standard MNIST dataset** is built into popular deep learning frameworks, including Keras, TensorFlow, PyTorch, etc. A sample of the MNIST 0–9 dataset can be seen in Figure 2.1 (*left*). The MNIST dataset will allow us to recognize the digits 0–9. Each of these digits is contained in a  $28 \times 28$  grayscale image.

But what about the letters A–Z? The standard MNIST dataset doesn't include examples of the characters A–Z — *how will we recognize them?*

**The answer is to use the National Institute of Standards and Technology (NIST) Special Database 19, which includes A–Z characters.** This dataset covers 62 American Standard Code for Information Interchange (ASCII) hexadecimal characters corresponding to the digits 0–9, capital letters A–Z, and lowercase letters a–z.

To make the dataset easier to use, Kaggle user Sachin Patel has released a simple comma-separated values (CSV) file: <http://pyimg.co/2psdb> [2]. This dataset takes the capital

letters A–Z from NIST Special Database 19 and rescales them from  $28 \times 28$  grayscale pixels to the same format as our MNIST data.



**Figure 2.1.** We are using *two separate datasets* to train our Keras/TensorFlow OCR model. On the *left*, we have the standard MNIST 0–9 dataset. On the *right*, we have the Kaggle A–Z dataset from Sachin Patel [2], which is based on the NIST Special Database 19 [3]

For this project, we will be using *just* the Kaggle A–Z dataset, which will make our pre-processing a breeze. A sample of it can be seen in Figure 2.1, (right).

We'll be implementing methods and utilities that will allow us to:

- i. Load *both* the datasets for MNIST 0–9 digits and Kaggle A–Z letters from disk
- ii. Combine these datasets into a *single*, unified character dataset
- iii. Handle class label skew/imbalance from having a different number of samples per character
- iv. Successfully train a Keras and TensorFlow model on the combined dataset
- v. Plot the results of the training and visualize the output of the validation data

To accomplish these tasks, let's move on to reviewing our directory structure for the project.

### 2.2.2 Project Structure

Let's get started reviewing our project directory structure:

---

```

|-- pyimagesearch
|   |-- __init__.py
|   |-- az_dataset
|   |   |-- __init__.py
|   |   |-- helpers.py
|   |-- models
|   |   |-- __init__.py
|   |   |-- resnet.py
|-- a_z_handwritten_data.zip
|-- a_z_handwritten_data.csv
|-- handwriting.model
|-- plot.png
|-- train_ocr_model.py

```

---

Inside our project directory, you'll find:

- `pyimagesearch` module: includes the submodules `az_dataset` for input/output (I/O) helper files and `models` for implementing the ResNet deep learning architecture
- `a_z_handwritten_data.csv`: contains the Kaggle A–Z dataset (you'll need to extract `a_z_handwritten_data.zip` to obtain this file)
- `handwriting.model`: where the deep learning ResNet model is saved
- `plot.png`: plots the results of the most recent run of training of ResNet
- `train_ocr_model.py`: the main driver file for training our ResNet model and displaying the results

Now that we have the lay of the land, let's dig into the I/O helper functions, we will use to load our digits and letters.

### 2.2.3 Implementing Our Dataset Loader Functions

To train our custom Keras and TensorFlow OCR model, we first need to implement two helper utilities that will allow us to load *both* the Kaggle A–Z datasets and the MNIST 0–9 digits from disk.

These I/O helper functions are appropriately named:

- `load_az_dataset`: for the Kaggle A–Z letters
- `load_mnist_dataset`: for the MNIST 0–9 digits

They can be found in the `helpers.py` file of `az_dataset` submodule of `pyimagesearch`.

Let's go ahead and examine this `helpers.py` file. We will begin with our import statements and then dig into our two helper functions: `load_az_dataset` and `load_mnist_dataset`:

---

```
1 # import the necessary packages
2 from tensorflow.keras.datasets import mnist
3 import numpy as np
```

---

**Line 2** imports the MNIST dataset, `mnist`, which is now one of the standard datasets that conveniently comes with Keras in `tensorflow.keras.datasets`.

Next, let's dive into `load_az_dataset`, the helper function to load the Kaggle A–Z letter data:

---

```
5 def load_az_dataset(datasetPath):
6     # initialize the list of data and labels
7     data = []
8     labels = []
9
10    # loop over the rows of the A-Z handwritten digit dataset
11    for row in open(datasetPath):
12        # parse the label and image from the row
13        row = row.split(",")
14        label = int(row[0])
15        image = np.array([int(x) for x in row[1:]], dtype="uint8")
16
17        # images are represented as single channel (grayscale) images
18        # that are 28x28=784 pixels -- we need to take this flattened
19        # 784-d list of numbers and reshape them into a 28x28 matrix
20        image = image.reshape(28, 28)
21
22        # update the list of data and labels
23        data.append(image)
24        labels.append(label)
```

---

Our function `load_az_dataset` takes a single argument `datasetPath`, which is the location of the Kaggle A–Z CSV file (**Line 5**). Then, we initialize our arrays to store the data and labels (**Lines 7 and 8**).

Each row in Sachin Patel's CSV file contains 785 columns — one column for the class label (i.e., “A–Z”) plus 784 columns corresponding to the  $28 \times 28$  grayscale pixels. Let's parse it.

Beginning on **Line 11**, we will loop over each row of our CSV file and parse out the label and the associated image. **Line 14** parses the label, which will be the integer label associated with a letter A–Z. For example, the letter “A” has a label corresponding to the integer “0,” and the letter “Z” has an integer label value of “25.”

Next, **Line 15** parses our image and casts it as a NumPy array of unsigned 8-bit integers, which correspond to each pixel's grayscale values from  $[0, 255]$ .

We reshape our image (**Line 20**) from a flat 784-d array to one that is  $28 \times 28$ , corresponding to the dimensions of each of our images.

We will then append each image and label to our data and label arrays, respectively (**Lines 23 and 24**).

To finish up this function, we will convert the data and labels to NumPy arrays and return the image data and labels:

---

```

26     # convert the data and labels to NumPy arrays
27     data = np.array(data, dtype="float32")
28     labels = np.array(labels, dtype="int")
29
30     # return a 2-tuple of the A-Z data and labels
31     return (data, labels)

```

---

Presently, our image data and labels are just Python lists, so we are going to typecast them as NumPy arrays of `float32` and `int`, respectively (**Lines 27 and 28**).

Great job implementing our first function!

Our next I/O helper function, `load_mnist_dataset`, is considerably simpler.

---

```

33 def load_mnist_dataset():
34     # load the MNIST dataset and stack the training data and testing
35     # data together (we'll create our own training and testing splits
36     # later in the project)
37     ((trainData, trainLabels), (testData, testLabels)) = mnist.load_data()
38     data = np.vstack([trainData, testData])
39     labels = np.hstack([trainLabels, testLabels])
40
41     # return a 2-tuple of the MNIST data and labels
42     return (data, labels)

```

---

**Line 37** loads our MNIST 0–9 digit data using Keras's helper function, `mnist.load_data`. Notice that we don't have to specify a `datasetPath` as we did for the Kaggle data because Keras, conveniently, has this dataset built-in.

Keras's `mnist.load_data` comes with a default split for training data, training labels, test data, and test labels. For now, we are just going to combine our training and test data for MNIST using `np.vstack` for our image data (**Line 38**) and `np.hstack` for our labels (**Line 39**).

Later, in `train_ocr_model.py`, we will be combining our MNIST 0–9 digit data with our Kaggle A–Z letters. At that point, we will create our custom split of test and training data.

Finally, **Line 42** returns the image data and associated labels to the calling function.

Congratulations! You have now completed the I/O helper functions to load both the digit and letter samples for OCR and deep learning. Next, we will examine our main driver file used for training and viewing the results.

#### 2.2.4 Implementing Our Training Script

In this section, we will train our OCR model using Keras, TensorFlow, and a PyImageSearch implementation of the popular and successful deep learning architecture, ResNet [4].

To get started, locate our primary driver file, `train_ocr_model.py`, found in the main project directory. This file contains a reference to a file `resnet.py`, located in the `models/` subdirectory under the `pyimagesearch` module.

Although we will not be doing a detailed walk-through of `resnet.py` in this chapter, you can get a feel for the ResNet architecture with my blog post on *Fine-Tuning ResNet with Keras and Deep Learning* (<http://pyimg.co/apodn> [5]). Please see my book, *Deep Learning for Computer Vision with Python* (<http://pyimg.co/dl4cv> [6]), for more advanced details.

Let's take a moment to review `train_ocr_model.py`. Afterward, we will come back and break it down, step by step.

First, we'll review the packages that we will import:

---

```
1 # set the matplotlib backend so figures can be saved in the background
2 import matplotlib
3 matplotlib.use("Agg")
4
5 # import the necessary packages
6 from pyimagesearch.models import ResNet
7 from pyimagesearch.az_dataset import load_mnist_dataset
8 from pyimagesearch.az_dataset import load_az_dataset
9 from tensorflow.keras.preprocessing.image import ImageDataGenerator
10 from tensorflow.keras.optimizers import SGD
11 from sklearn.preprocessing import LabelBinarizer
12 from sklearn.model_selection import train_test_split
13 from sklearn.metrics import classification_report
14 from imutils import build_montages
15 import matplotlib.pyplot as plt
16 import numpy as np
17 import argparse
18 import cv2
```

---

This is a long list of import statements, but don't worry. It means we have many packages that have already been written to make our lives much easier.

Starting on **Line 2**, we will import `matplotlib` and set up the back end of it by writing the results to a file using `matplotlib.use("Agg")` (**Line 3**).

We then have some imports from our custom `pyimagesearch` module for our deep learning architecture and our I/O helper functions that we just reviewed:

- We import `ResNet` from our `pyimagesearch.model`, which contains our custom implementation of the popular ResNet deep learning architecture (**Line 6**).
- Next, we import our I/O helper functions `load_mnist_data` (**Line 7**) and `load_az_dataset` (**Line 8**) from `pyimagesearch.az_dataset`.

We have a couple of imports from the `Keras` module of `TensorFlow`, which greatly simplify our data augmentation and training:

- **Line 9** imports `ImageDataGenerator` to help us efficiently augment our dataset.
- We then import `SGD`, the stochastic gradient descent (SGD) optimization algorithm (**Line 10**).

Following on, we import three helper functions from `scikit-learn` to help us label our data, split our testing and training datasets, and print out a classification report to show us our results:

- To convert our labels from integers to a vector in what is called one-hot encoding, we import `LabelBinarizer` (**Line 11**).
- To help us easily split out our testing and training datasets, we import `train_test_split` from `scikit-learn` (**Line 12**).
- From the `metrics` submodule, we import `classification_report` to print out a nicely formatted classification report (**Line 13**).

From `imutils`, we import `build_montages` to help us build a montage from a list of images (**Line 14**). For more information on creating montages, please refer to my *Montages with OpenCV* tutorial (<http://pyimg.co/vquhs> [7]).

We will round out our notable imports with `matplotlib` (**Line 18**) and `OpenCV` (**Line 21**).

Now, let's review our three command line arguments:

---

20 # construct the argument parser and parse the arguments  
21 ap = argparse.ArgumentParser()

---

```

22 ap.add_argument("-a", "--az", required=True,
23     help="path to A-Z dataset")
24 ap.add_argument("-m", "--model", type=str, required=True,
25     help="path to output trained handwriting recognition model")
26 ap.add_argument("-p", "--plot", type=str, default="plot.png",
27     help="path to output training history file")
28 args = vars(ap.parse_args())

```

---

We have three arguments to review:

- **--az**: The path to the Kaggle A–Z dataset
- **--model**: The path to output the trained handwriting recognition model
- **--plot**: The path to output the training history file

So far, we have our imports, convenience function, and command line `args` ready to go. We have several steps remaining to set up the training for ResNet, compile it, and train it.

Now, we will set up the training parameters for ResNet and load our digit and letter data using the helper functions that we already reviewed:

---

```

30 # initialize the number of epochs to train for, initial learning rate,
31 # and batch size
32 EPOCHS = 50
33 INIT_LR = 1e-1
34 BS = 128
35
36 # load the A-Z and MNIST datasets, respectively
37 print("[INFO] loading datasets...")
38 (azData, azLabels) = load_az_dataset(args["az"])
39 (digitsData, digitsLabels) = load_mnist_dataset()

```

---

**Lines 32–34** initialize the parameters for the training of our ResNet model.

Then, we load the data and labels for the Kaggle A–Z and MNIST 0–9 digits data, respectively (**Lines 38 and 39**), using the I/O helper functions that we reviewed at the beginning of the post.

Next, we are going to perform several steps to prepare our data and labels to be compatible with our ResNet deep learning model in Keras and TensorFlow:

---

```

41 # the MNIST dataset occupies the labels 0–9, so let's add 10 to every
42 # A-Z label to ensure the A-Z characters are not incorrectly labeled
43 # as digits
44 azLabels += 10

```

---

```

45
46 # stack the A-Z data and labels with the MNIST digits data and labels
47 data = np.vstack([azData, digitsData])
48 labels = np.hstack([azLabels, digitsLabels])
49
50 # each image in the A-Z and MNIST digits datasets are 28x28 pixels;
51 # however, the architecture we're using is designed for 32x32 images,
52 # so we need to resize them to 32x32
53 data = [cv2.resize(image, (32, 32)) for image in data]
54 data = np.array(data, dtype="float32")
55
56 # add a channel dimension to every image in the dataset and scale the
57 # pixel intensities of the images from [0, 255] down to [0, 1]
58 data = np.expand_dims(data, axis=-1)
59 data /= 255.0

```

---

As we combine our letters and numbers into a single character dataset, we want to remove any ambiguity where there is overlap in the labels so that each label in the combined character set is unique.

Currently, our labels for *A*–*Z* go from  $[0, 25]$ , corresponding to each letter of the alphabet. The labels for our digits go from  $0$ – $9$ , so there is overlap — which would be problematic if we were to combine them directly.

No problem! There is a straightforward fix. We will add ten to all of our *A*–*Z* labels, so they all have integer label values greater than our digit label values (**Line 44**). We have a unified labeling schema for digits  $0$ – $9$  and letters *A*–*Z* without any overlap in the labels' values.

**Line 47** combines our datasets for our digits and letters into a single character dataset using `np.vstack`. Likewise, **Line 51** unifies our corresponding labels for our digits and letters using `np.hstack`.

Our ResNet architecture requires the images to have input dimensions of  $32 \times 32$ , but our input images currently have a size of  $28 \times 28$ . We resize each of the images using `cv2.resize` (**Line 53**).

We have two final steps to prepare our data for use with ResNet. On **Line 58**, we will add an extra “channel” dimension to every image in the dataset to make it compatible with the ResNet model in Keras/TensorFlow. Finally, we will scale our pixel intensities from a range of  $[0, 255]$  down to  $[0.0, 1.0]$  (**Line 59**).

Our next step is to prepare the labels for ResNet, weight the labels to account for the skew in the number of times each class (character) is represented in the data, and partition the data into test and training splits:

---

```

61 # convert the labels from integers to vectors
62 le = LabelBinarizer()

```

---

```

63 labels = le.fit_transform(labels)
64 counts = labels.sum(axis=0)
65
66 # account for skew in the labeled data
67 classTotals = labels.sum(axis=0)
68 classWeight = {}
69
70 # loop over all classes and calculate the class weight
71 for i in range(0, len(classTotals)):
72     classWeight[i] = classTotals.max() / classTotals[i]
73
74 # partition the data into training and testing splits using 80% of
75 # the data for training and the remaining 20% for testing
76 (trainX, testX, trainY, testY) = train_test_split(data,
77     labels, test_size=0.20, stratify=labels, random_state=42)

```

---

We instantiate a LabelBinarizer (**Line 62**), and then we convert the labels from integers to a vector of binaries with one-hot encoding (**Line 63**) using `le.fit_transform`.

**Lines 67–72** weight each class, based on the frequency of occurrence of each character.

Next, we will use the scikit-learn `train_test_split` utility (**Lines 76 and 77**) to partition the data into 80% training and 20% testing.

From there, we'll augment our data using an image generator from Keras:

---

```

79 # construct the image generator for data augmentation
80 aug = ImageDataGenerator(
81     rotation_range=10,
82     zoom_range=0.05,
83     width_shift_range=0.1,
84     height_shift_range=0.1,
85     shear_range=0.15,
86     horizontal_flip=False,
87     fill_mode="nearest")

```

---

We can improve our ResNet classifier's results by augmenting the input data for training using an `ImageDataGenerator`. **Lines 80–87** include various rotations, scaling the size, horizontal translations, vertical translations, and tilts in the images. For more details on data augmentation, see both my tutorial on *Keras ImageDataGenerator and Data Augmentation* (<http://pyimg.co/pedyk> [8]) as well as my book, *Deep Learning for Computer Vision with Python* (<http://pyimg.co/dl4cv> [6]).

Now we are ready to initialize and compile the ResNet network:

---

```

89 # initialize and compile our deep neural network
90 print("[INFO] compiling model...")

```

---

---

```

91 opt = SGD(lr=INIT_LR, decay=INIT_LR / EPOCHS)
92 model = ResNet.build(32, 32, 1, len(le.classes_), (3, 3, 3),
93                      (64, 64, 128, 256), reg=0.0005)
94 model.compile(loss="categorical_crossentropy", optimizer=opt,
95                  metrics=["accuracy"])

```

---

Using the `SGD` optimizer and a standard learning rate decay schedule, we build our ResNet architecture (**Lines 91–93**). Each character/digit is represented as a  $32 \times 32$  pixel grayscale image as is evident by the first three parameters to ResNet's `build` method.

**Lines 94 and 95** compile our model with “`categorical_crossentropy`” loss and our established `SGD` optimizer. Please beware that if you are working with a 2-class only dataset (we are not), you would need to use the “`binary_crossentropy`” loss function.

Next, we will train the network, define label names, and evaluate the performance of the network:

---

```

97 # train the network
98 print("[INFO] training network...")
99 H = model.fit(
100     aug.flow(trainX, trainY, batch_size=BS),
101     validation_data=(testX, testY),
102     steps_per_epoch=len(trainX) // BS,
103     epochs=EPOCHS,
104     class_weight=classWeight,
105     verbose=1)
106
107 # define the list of label names
108 labelNames = "0123456789"
109 labelNames += "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
110 labelNames = [l for l in labelNames]
111
112 # evaluate the network
113 print("[INFO] evaluating network...")
114 predictions = model.predict(testX, batch_size=BS)
115 print(classification_report(testY.argmax(axis=1),
116                             predictions.argmax(axis=1), target_names=labelNames))

```

---

We train our model using the `model.fit` method (**Lines 99–105**). The parameters are as follows:

- `aug.flow`: establishes in-line data augmentation
- `validation_data`: test input images (`testX`) and test labels (`testY`)
- `steps_per_epoch`: how many batches are run per each pass of the full training data
- `epochs`: the number of complete passes through the entire dataset during training

- `class_weight`: weights due to the imbalance of data samples for various classes (e.g., digits and letters) in the training data
- `verbose`: shows a progress bar during the training

Next, we establish labels for each character. **Lines 108–110** concatenates all of our digits and letters and form an array where each member of the array is a single digit or number.

To evaluate our model, we make predictions on the test set and print our classification report. We'll see the report very soon in the next section!

**Lines 115 and 116** print out the results using the convenient scikit-learn `classification_report` utility.

We will save the model to disk, plot the results of the training history, and save the training history:

---

```

118 # save the model to disk
119 print("[INFO] serializing network...")
120 model.save(args["model"], save_format="h5")
121
122 # construct a plot that plots and saves the training history
123 N = np.arange(0, EPOCHS)
124 plt.style.use("ggplot")
125 plt.figure()
126 plt.plot(N, H.history["loss"], label="train_loss")
127 plt.plot(N, H.history["val_loss"], label="val_loss")
128 plt.title("Training Loss and Accuracy")
129 plt.xlabel("Epoch #")
130 plt.ylabel("Loss/Accuracy")
131 plt.legend(loc="lower left")
132 plt.savefig(args["plot"])

```

---

As we have finished our training, we need to save the model comprised of the architecture and final weights. We will save our model to disk, as a Hierarchical Data Format version 5 (HDF5) file, which is specified by the `save_format` (**Line 120**).

Next, we use matplotlib to generate a line plot for the training loss and validation set loss along with titles, labels for the axes, and a legend. The data for the training and validation losses come from the history of `H`, which are the results of `model.fit` from above with one point for every epoch (**Lines 123–131**). The plot of the training loss curves is saved to `plot.png` (**Line 132**).

Finally, let's code our visualization procedure, so we can see whether our model is working or not:

---

```

134 # initialize our list of output images

```

---

```

135 images = []
136
137 # randomly select a few testing characters
138 for i in np.random.choice(np.arange(0, len(testY)), size=(49, )):
139     # classify the character
140     probs = model.predict(testX[np.newaxis, i])
141     prediction = probs.argmax(axis=1)
142     label = labelNames[prediction[0]]
143
144     # extract the image from the test data and initialize the text
145     # label color as green (correct)
146     image = (testX[i] * 255).astype("uint8")
147     color = (0, 255, 0)
148
149     # otherwise, the class label prediction is incorrect
150     if prediction[0] != np.argmax(testY[i]):
151         color = (0, 0, 255)
152
153     # merge the channels into one image, resize the image from 32x32
154     # to 96x96 so we can better see it and then draw the predicted
155     # label on the image
156     image = cv2.merge([image] * 3)
157     image = cv2.resize(image, (96, 96), interpolation=cv2.INTER_LINEAR)
158     cv2.putText(image, label, (5, 20), cv2.FONT_HERSHEY_SIMPLEX, 0.75,
159                 color, 2)
160
161     # add the image to our list of output images
162     images.append(image)

```

---

**Line 135** initializes our array of test images. Starting on **Line 138**, we randomly select 49 characters (to form a  $7 \times 7$  grid) and proceed to:

- Classify the character using our ResNet-based model (**Lines 140–142**)
- Grab the individual character `image` from our test data (**Line 146**)
- Set an annotation text `color` as green (correct) or red (incorrect) via **Lines 147–151**
- Create an RGB representation of our single channel `image` and `resize` it for inclusion in our visualization montage (**Lines 156 and 157**)
- Annotate the colored text label (**Lines 158 and 159**)
- Add the `image` to our output `images` array (**Line 162**)

We're almost done. Only one code block left to go:

---

```

164 # construct the montage for the images
165 montage = build_montages(images, (96, 96), (7, 7))[0]
166

```

---

```
167 # show the output montage
168 cv2.imshow("OCR Results", montage)
169 cv2.waitKey(0)
```

---

To close out, we assemble each annotated character image into an OpenCV montage visualization grid (<http://pyimg.co/vquhs> [7]), displaying the result until a key is pressed (**Lines 165–169**).

Congratulations! We learned a lot along the way! Next, we'll see the results of our hard work.

### 2.2.5 Training Our Keras and TensorFlow OCR Model

Before we run our training script, take a second to recall from the last section that our script:

- i. Loads MNIST 0–9 digits and Kaggle A–Z letters
- ii. Trains a ResNet model on the dataset
- iii. Produces a visualization so that we can ensure it is working properly

In this section, we'll execute our OCR model training and visualization script. Let's do that now:

---

```
$ python train_ocr_model.py --az a_z_handwritten_data.csv --model handwriting.model
[INFO] loading datasets...
[INFO] compiling model...
[INFO] training network...
Epoch 1/50
2765/2765 [=====] - 102s 37ms/step - loss: 0.9246 - accuracy: 0.8258
↳ val_loss: 0.4715 - val_accuracy: 0.9381
Epoch 2/50
2765/2765 [=====] - 96s 35ms/step - loss: 0.4708 - accuracy: 0.9374 -
↳ val_loss: 0.4146 - val_accuracy: 0.9521
Epoch 3/50
2765/2765 [=====] - 95s 34ms/step - loss: 0.4349 - accuracy: 0.9454 -
↳ val_loss: 0.3986 - val_accuracy: 0.9543
...
Epoch 48/50
2765/2765 [=====] - 94s 34ms/step - loss: 0.3501 - accuracy: 0.9609 -
↳ val_loss: 0.3442 - val_accuracy: 0.9624
Epoch 49/50
2765/2765 [=====] - 94s 34ms/step - loss: 0.3496 - accuracy: 0.9606 -
↳ val_loss: 0.3444 - val_accuracy: 0.9624
Epoch 50/50
2765/2765 [=====] - 94s 34ms/step - loss: 0.3494 - accuracy: 0.9608 -
↳ val_loss: 0.3468 - val_accuracy: 0.9616
[INFO] evaluating network...
      precision    recall   f1-score   support
          0       0.48      0.44      0.46      1381
          1       0.98      0.97      0.97      1575
          2       0.90      0.94      0.92      1398
```

3	0.97	0.99	0.98	1428
4	0.92	0.96	0.94	1365
5	0.80	0.89	0.85	1263
6	0.96	0.97	0.96	1375
7	0.95	0.98	0.97	1459
8	0.97	0.97	0.97	1365
9	0.98	0.98	0.98	1392
A	0.99	0.99	0.99	2774
B	0.97	0.98	0.98	1734
C	0.99	0.99	0.99	4682
D	0.96	0.96	0.96	2027
E	0.99	0.99	0.99	2288
F	0.99	0.95	0.97	232
G	0.96	0.94	0.95	1152
H	0.96	0.96	0.96	1444
I	0.98	0.96	0.97	224
J	0.98	0.95	0.97	1699
K	0.98	0.96	0.97	1121
L	0.97	0.98	0.98	2317
M	0.99	0.99	0.99	2467
N	0.99	0.99	0.99	3802
O	0.93	0.94	0.93	11565
P	1.00	0.99	0.99	3868
Q	0.97	0.96	0.96	1162
R	0.98	0.99	0.98	2313
S	0.98	0.97	0.98	9684
T	0.99	0.99	0.99	4499
U	0.98	0.99	0.99	5802
V	0.98	0.98	0.98	836
W	0.99	0.98	0.98	2157
X	0.99	0.98	0.98	1254
Y	0.98	0.94	0.96	2172
Z	0.94	0.92	0.93	1215
accuracy			0.96	88491
macro avg	0.95	0.95	0.95	88491
weighted avg	0.96	0.96	0.96	88491

[INFO] serializing network...

**Remark 2.1.** When loaded into memory, the entire A–Z and 0–9 character datasets consume over 4GB of RAM. If you are using the VM included with your purchase of this book, then you will need to adjust the VM settings to include more RAM (4GB is the default amount). Setting the default RAM allocation to 8GB on the VM is more than sufficient.

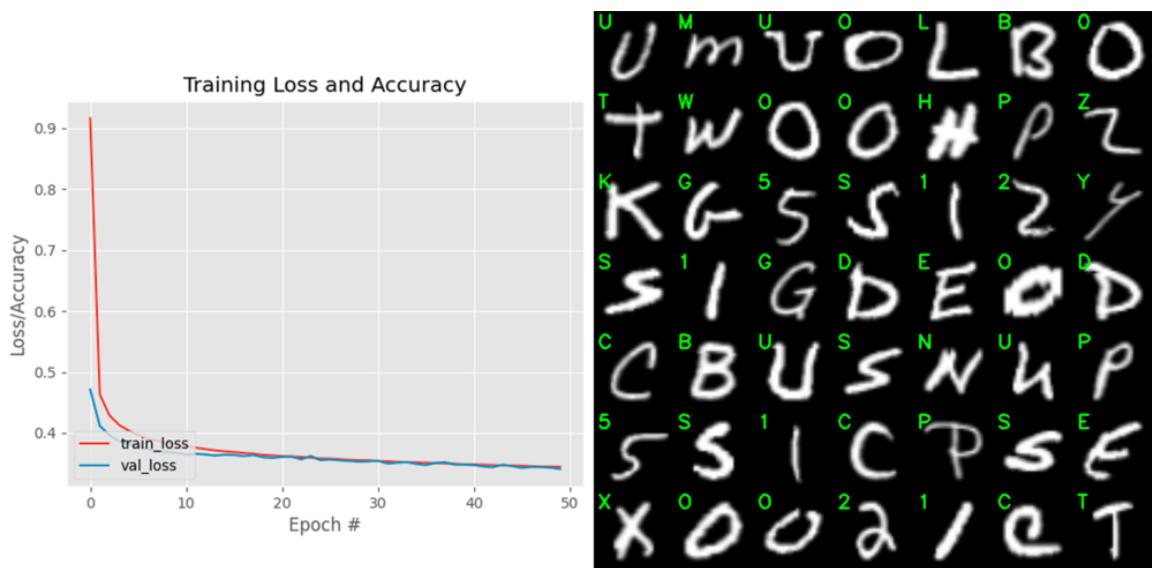
As you can see, our Keras/TensorFlow OCR model is obtaining **≈96% accuracy** on the testing set.

The training history plot can be seen in Figure 2.2 (*left*). As evidenced by the plot, there are few overfitting signs, implying that our Keras and TensorFlow model performs well at our basic OCR task.

We can see the prediction output from a sample of our testing set in Figure 2.2 (*right*). As our results show, our handwriting recognition model is performing quite well!

And finally, if you check your current working directory, you should find a new file named `handwriting.model`:

```
$ ls *.model  
handwriting.model
```



**Figure 2.2.** *Left:* A plot of our training history. Our model shows little signs of overfitting, implying that our OCR model is performing well. *Right:* A sample output of character classifications made on the testing set by our trained OCR model. Note that we have correctly classified each character.

This file is our serialized Keras and TensorFlow OCR model — we'll be using it in Chapter 3 on handwriting recognition.

## 2.3 Summary

In this chapter, you learned how to train a custom OCR model using Keras and TensorFlow.

Our model was trained to recognize alphanumeric characters, including the **digits 0–9** and the **letters A–Z**. Overall, our Keras and TensorFlow OCR model obtained  $\approx 96\%$  **accuracy** on our testing set.

In the next chapter, you'll learn how to take our trained Keras/TensorFlow OCR model and use it for handwriting recognition on custom input images.



## Chapter 3

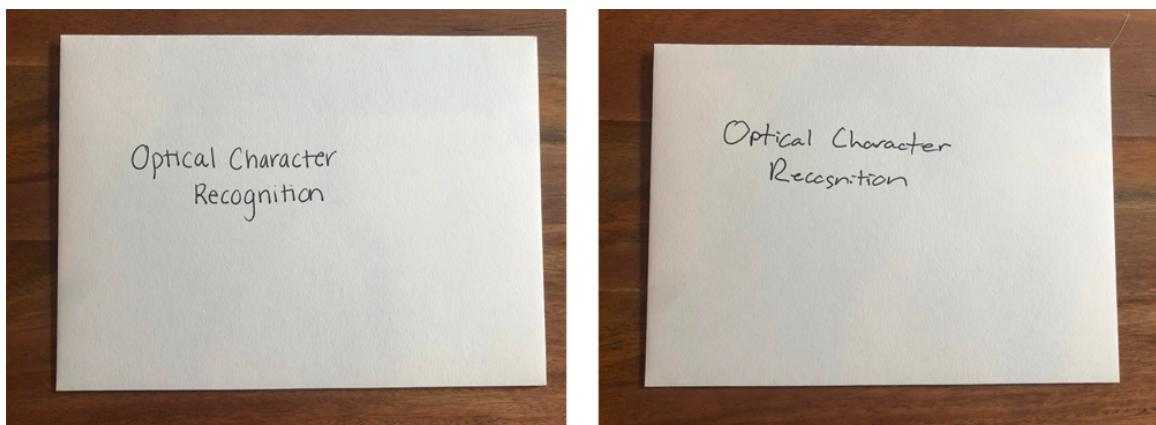
# Handwriting Recognition with Keras and TensorFlow

In this chapter, you will learn how to perform optical character recognition (OCR) handwriting recognition using OpenCV, Keras, and TensorFlow.

As you'll see further below, **handwriting recognition tends to be *significantly harder* than traditional OCR that uses specific fonts/characters**. This concept is so challenging because there are nearly *infinite variations* of handwriting styles, unlike computer fonts. Every one of us has a personal style that is *specific and unique*.

My wife, for example, has *amazing* penmanship. Her handwriting is not only legible, but it's stylized in a way that you would think a professional calligrapher wrote it (Figure 3.1, *left*).

Me, on the other hand, my handwriting looks like someone crossed a doctor with a deranged squirrel (Figure 3.1, *right*).



**Figure 3.1.** My wife has beautiful penmanship (*left*), while my handwriting looks like a fourth grader (*right*).

It's *barely* legible. I'm often asked by those who read my handwriting at least 2–3 clarifying what a specific word or phrase is. And on more than one occasion, I've had to admit that I couldn't read them either.

Talk about embarrassing! Truly, it's a wonder they ever let me out of grade school.

These handwriting style variations pose a problem for OCR engines, typically trained on computer fonts, *not* handwriting fonts.

And worse, handwriting recognition is further complicated because letters can “connect” and “touch” each other, making it incredibly challenging for OCR algorithms to separate them, ultimately leading to incorrect OCR results.

Handwriting recognition is arguably the “holy grail” of OCR. We’re not there yet, but with the help of deep learning, we’re making *tremendous* strides.

This chapter will serve as an introduction to handwriting recognition. You’ll see examples of where handwriting recognition has performed well and other instances where it has failed to correctly OCR a handwritten character. I genuinely think you’ll find value in reading the rest of this handwriting recognition guide.

## 3.1 Chapter Learning Objectives

In this chapter, you will:

- Discover what handwriting recognition is
- Learn how to take our handwriting recognition model (trained in the previous chapter) and use it to make predictions
- Create a Python script that loads the input model and a sample image, pre-process it, and then performs handwriting recognition

## 3.2 OCR’ing Handwriting with Keras and TensorFlow

In the first part of this chapter, we’ll discuss handwriting recognition and how it is different from “traditional” OCR.

I’ll then provide a brief review of the process for training our recognition model using Keras and TensorFlow — we’ll be using this trained model to OCR handwriting in this chapter.

We’ll review our project structure and then implement a Python script to perform handwriting recognition with OpenCV, Keras, and TensorFlow.

To wrap up this OCR section, we'll discuss our handwriting recognition results, including what worked and what didn't.

### 3.2.1 What Is Handwriting Recognition?

Traditional OCR algorithms and techniques assume we're working with a fixed font of some sort. In the early 1900s, that could have been the font used by microfilms.

In the 1970s, specialized fonts were *explicitly* developed for OCR algorithms, thereby making them more accurate. By the 2000s, we could use the fonts pre-installed on our computers to automatically generate training data and use these fonts to train our OCR models.

Each of these fonts had something in common:

- i. They were engineered in some manner.
- ii. There was a *predictable* and *assumed* space between each character (thereby making segmentation easier).
- iii. The styles of the fonts were more conducive to OCR.

Essentially, engineered/computer-generated fonts make OCR *far easier*. Figure 3.2 shows how computer text on the *right* is more uniform and segmented than handwritten text which is on the *left*.



Figure 3.2. Left: Handwritten text. Right: Typed text.

Handwriting recognition is an entirely different beast, though. Consider the extreme amounts of variation and how characters often overlap. Everyone has a unique writing style.

Characters can be elongated, swooped, slanted, stylized, crunched, connected, tiny, gigantic, etc. (and come in any of these combinations).

Digitizing handwriting recognition is *exceptionally* challenging and is still *far* from solved — but deep learning helps us improve our handwriting recognition accuracy.

### 3.2.2 Project Structure

Let's get started by reviewing our project directory structure:

---

```
|-- images
|   |-- hello_world.png
|   |-- umbc_address.png
|   |-- umbc_zipcode.png
|-- handwriting.model
|-- ocr_handwriting.py
```

---

The project directory contains:

- `handwriting.model`: The custom OCR ResNet model we trained in the previous chapter
- `images/` subdirectory: Contains three portable network graphic (PNG) test files for us to OCR with our Python driver script
- `ocr_handwriting.py`: The main Python script that we will use to OCR our handwriting samples

Now that we have a handle on the project structure let's dive into our driver script.

### 3.2.3 Implementing Our Handwriting Recognition Script

Let's open `ocr_handwriting.py` and review it, starting with the imports and command line arguments:

---

```
1 # import the necessary packages
2 from tensorflow.keras.models import load_model
3 from imutils.contours import sort_contours
4 import numpy as np
5 import argparse
6 import imutils
7 import cv2
8
9 # construct the argument parser and parse the arguments
10 ap = argparse.ArgumentParser()
11 ap.add_argument("-i", "--image", required=True,
12     help="path to input image")
13 ap.add_argument("-m", "--model", type=str, required=True,
```

---

```
14     help="path to trained handwriting recognition model")
15 args = vars(ap.parse_args())
```

---

**Line 2** imports the `load_model` utility, which allows us to easily load the OCR model that we developed in the previous chapter.

Using my `imutils` package, we then import `sort_contours` (**Line 3**) and `imutils` (**Line 6**) to facilitate operations with contours and resizing images.

Our command line arguments include `--image`, our input image, and `--model`, the path to our trained handwriting recognition model.

Next, we will load our custom handwriting OCR model that we trained earlier in this book:

---

```
17 # load the handwriting OCR model
18 print("[INFO] loading handwriting OCR model...")
19 model = load_model(args["model"])
```

---

The `load_model` utility from Keras and TensorFlow makes it super simple to load our serialized handwriting recognition model (**Line 19**). Recall that our OCR model uses the ResNet deep learning architecture to classify each character corresponding to a digit 0–9 or a letter A–Z.

Since we've loaded our model from disk, let's grab our image, pre-process it, and find character contours:

---

```
21 # load the input image from disk, convert it to grayscale, and blur
22 # it to reduce noise
23 image = cv2.imread(args["image"])
24 gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
25 blurred = cv2.GaussianBlur(gray, (5, 5), 0)
26
27 # perform edge detection, find contours in the edge map, and sort the
28 # resulting contours from left-to-right
29 edged = cv2.Canny(blurred, 30, 150)
30 cnts = cv2.findContours(edged.copy(), cv2.RETR_EXTERNAL,
31                         cv2.CHAIN_APPROX_SIMPLE)
32 cnts = imutils.grab_contours(cnts)
33 cnts = sort_contours(cnts, method="left-to-right") [0]
34
35 # initialize the list of contour bounding boxes and associated
36 # characters that we'll be OCR'ing
37 chars = []
```

---

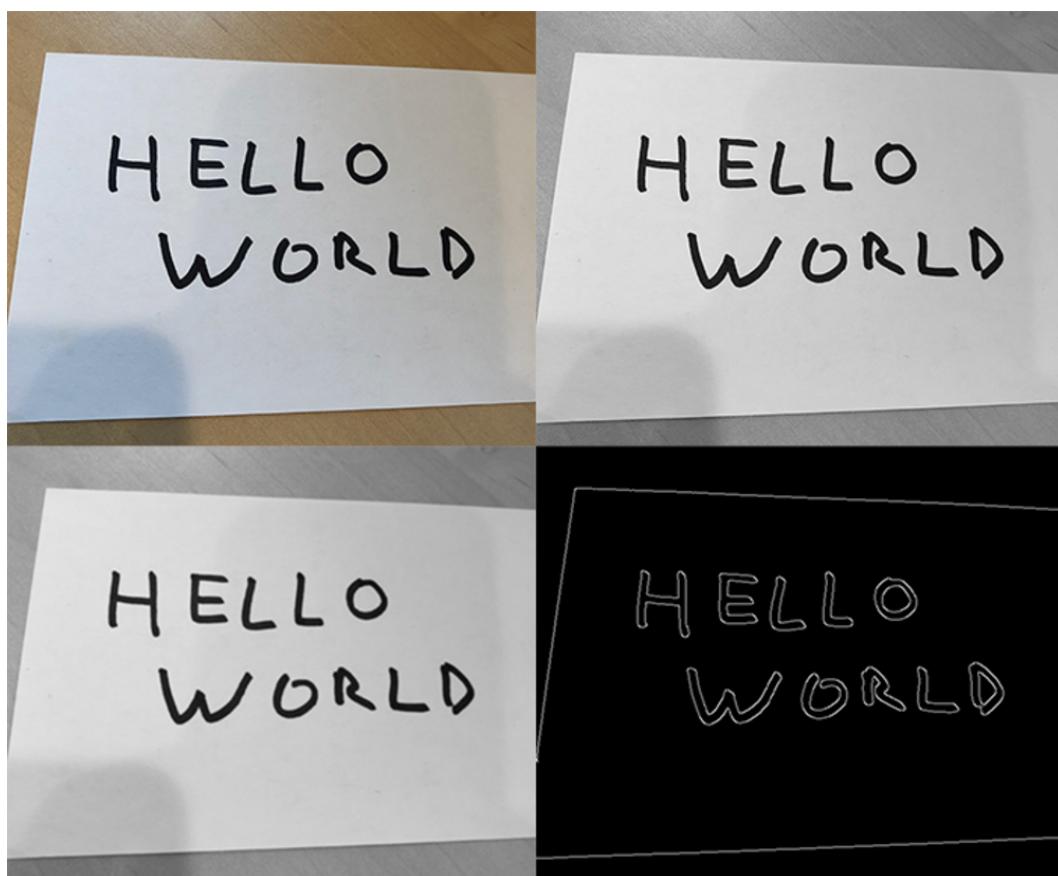
After loading the image (**Line 23**), we convert it to grayscale (**Line 24**), and then apply Gaussian blurring to reduce noise (**Line 25**).

From there, we detect the edges of our blurred image using `cv2.Canny` (**Line 29**).

To locate the contours for each character, we apply contour detection (**Lines 30 and 31**). To conveniently sort the contours from "left-to-right" (**Line 33**), we use my `sort_contours` method.

**Line 37** initializes the `chars` list, which will soon hold each character image and associated bounding box.

In Figure 3.3, we can see the sample results from our image pre-processing steps.



**Figure 3.3.** Steps of our handwriting OCR pipeline. We have our original color image (upper-left), our grayscale image (upper-right), our blurred image with reduced noise (lower-left), and our edge-detection map (lower-right).

Our next step will involve a large contour processing loop. Let's break that down in more detail so that it is easier to get through:

---

```

39 # loop over the contours
40 for c in cnts:
41     # compute the bounding box of the contour
42     (x, y, w, h) = cv2.boundingRect(c)

```

```

43
44     # filter out bounding boxes, ensuring they are neither too small
45     # nor too large
46     if (w >= 5 and w <= 150) and (h >= 15 and h <= 120):
47         # extract the character and threshold it to make the character
48         # appear as *white* (foreground) on a *black* background, then
49         # grab the width and height of the thresholded image
50         roi = gray[y:y + h, x:x + w]
51         thresh = cv2.threshold(roi, 0, 255,
52             cv2.THRESH_BINARY_INV | cv2.THRESH_OTSU)[1]
53         (tH, tW) = thresh.shape
54
55         # if the width is greater than the height, resize along the
56         # width dimension
57         if tW > tH:
58             thresh = imutils.resize(thresh, width=32)
59
60         # otherwise, resize along the height
61     else:
62         thresh = imutils.resize(thresh, height=32)

```

---

Beginning on **Line 40**, we loop over each contour and perform a series of four steps.

In **Step #1**, we select appropriately sized contours and extract them by:

- Computing the bounding box of the contour (**Line 42**)
- Making sure the bounding boxes are of reasonable size, filtering out those that are either too large or too small (**Line 46**)
- For each bounding box meeting our size criteria, we extract the region of interest (`roi`) associated with the character (**Line 50**)

Next, we move on to **Step #2** of our algorithm — cleaning up the images using Otsu's thresholding technique, with the ultimate goal of segmenting the characters such that they appear as *white* on a *black* background (**Lines 51 and 52**).

We then arrive at **Step #3**, where the ultimate goal is to resize every character to  $32 \times 32$  pixels while maintaining the aspect ratio.

Depending on whether the width is greater than the height or the height is greater than the width, we resize the thresholded character ROI accordingly (**Lines 57–62**).

But wait! Before we can continue our loop that began on **Line 40**, we need to pad these ROIs and add them to the `chars` list:

---

```

64     # re-grab the image dimensions (now that its been resized)
65     # and then determine how much we need to pad the width and

```

```

66     # height such that our image will be 32x32
67     (tH, tW) = thresh.shape
68     dX = int(max(0, 32 - tW) / 2.0)
69     dY = int(max(0, 32 - tH) / 2.0)
70
71     # pad the image and force 32x32 dimensions
72     padded = cv2.copyMakeBorder(thresh, top=dY, bottom=dY,
73         left=dX, right=dX, borderType=cv2.BORDER_CONSTANT,
74         value=(0, 0, 0))
75     padded = cv2.resize(padded, (32, 32))
76
77     # prepare the padded image for classification via our
78     # handwriting OCR model
79     padded = padded.astype("float32") / 255.0
80     padded = np.expand_dims(padded, axis=-1)
81
82     # update our list of characters that will be OCR'd
83     chars.append((padded, (x, y, w, h)))

```

---

Now that we have padded the ROIs and added them to the `chars` list, we can finish resizing and padding by computing the necessary delta padding in the *x* and *y* directions. (**Lines 67–69**) , followed by applying the padding to create the `padded` image (**Lines 72–74**). Due to integer rounding on **Lines 68 and 69**, we then explicitly resize the image to 32 x 32 pixels on **Line 75** to round out the pre-processing phase.

Ultimately, these pre-processing steps ensure that each image is 32 x 32 while the aspect ratio is correctly maintained.

Next comes **Step #4**, where we prepare each padded ROI for classification as a character:

- Scale pixel intensities to the range [0, 1] and add a batch dimension (**Lines 79 and 80**).
- Package both the padded character and bounding box as a 2-tuple, and add it to our `chars` list (**Line 83**).

With our extracted and prepared set of character ROIs completed, we can perform handwriting recognition:

---

```

85     # extract the bounding box locations and padded characters
86     boxes = [b[1] for b in chars]
87     chars = np.array([c[0] for c in chars], dtype="float32")
88
89     # OCR the characters using our handwriting recognition model
90     preds = model.predict(chars)
91
92     # define the list of label names
93     labelNames = "0123456789"

```

---

```

94     labelNames += "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
95     labelNames = [l for l in labelNames]
```

---

**Lines 86 and 87** extract the original bounding boxes with associated chars in NumPy array format.

To perform handwriting recognition OCR on our set of pre-processed characters, we classify the entire batch with the `model.predict` method (**Line 90**). This results in a list of predictions, `preds`.

As we learned from our previous chapter, we then concatenate our labels for our digits and letters into a single list of `labelNames` (**Lines 93–95**).

We're almost done! It's time to see the fruits of our labor. To see if our handwriting recognition results meet our expectations, let's visualize and display them:

---

```

97     # loop over the predictions and bounding box locations together
98     for (pred, (x, y, w, h)) in zip(preds, boxes):
99         # find the index of the label with the largest corresponding
100        # probability, then extract the probability and label
101        i = np.argmax(pred)
102        prob = pred[i]
103        label = labelNames[i]
104
105        # draw the prediction on the image
106        print("[INFO] {} - {:.2f}%".format(label, prob * 100))
107        cv2.rectangle(image, (x, y), (x + w, y + h), (0, 255, 0), 2)
108        cv2.putText(image, label, (x - 10, y - 10),
109                    cv2.FONT_HERSHEY_SIMPLEX, 1.2, (0, 255, 0), 2)
110
111        # show the image
112        cv2.imshow("Image", image)
113        cv2.waitKey(0)
```

---

Wrapping up, we loop over each prediction and corresponding bounding box (**Line 98**).

Inside the loop, we grab the highest probability prediction resulting in the particular character's label (**Lines 101–103**).

To visualize the results, we annotate each character with the bounding box and `label` text and display the result (**Lines 107–113**). To cycle to the next character, press any key.

Congratulations! You have completed the main Python driver file to perform OCR on input images. Let's take a look at our results in the next section.

### 3.2.4 Handwriting Recognition Results

Let's put our handwriting recognition model to work! Open a terminal and execute the following command:

---

```
$ python ocr_handwriting.py --model handwriting.model --image
→ images/hello_world.png
[INFO] loading handwriting OCR model...
[INFO] H - 92.48%
[INFO] W - 54.50%
[INFO] E - 94.93%
[INFO] L - 97.58%
[INFO] 2 - 65.73%
[INFO] L - 96.56%
[INFO] R - 97.31%
[INFO] 0 - 37.92%
[INFO] L - 97.13%
[INFO] D - 97.83%
```

---

In Figure 3.4, we are attempting to OCR the handwritten text “Hello World.” Our handwriting recognition model performed well here, but made **two mistakes**:

- i. First, it confused the letter “O” with the digit “0” (zero) — that’s an understandable mistake.
- ii. Second, and a bit more concerning, the handwriting recognition model confused the “O” in “World” with a “2.”

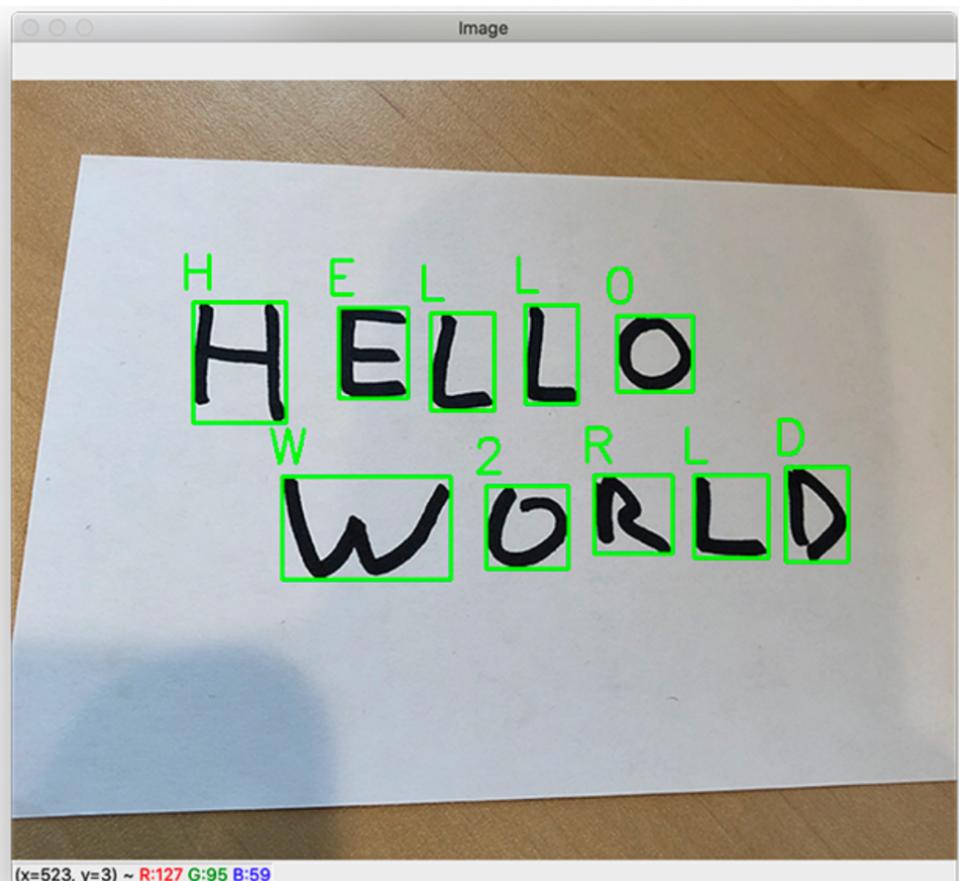
That’s a combination of a pre-processing issue along with not enough real-world training data for the “2” and “W” characters.

This next example contains the handwritten name and ZIP code of my *alma mater*, University of Maryland, Baltimore County (UMBC):

---

```
$ python ocr_handwriting.py --model handwriting.model --image
→ images/umbc_zipcode.png
[INFO] loading handwriting OCR model...
[INFO] U - 34.76%
[INFO] 2 - 97.88%
[INFO] M - 75.04%
[INFO] 7 - 51.22%
[INFO] B - 98.63%
[INFO] 2 - 99.35%
[INFO] C - 63.28%
[INFO] 5 - 66.17%
[INFO] 0 - 66.34%
```

---



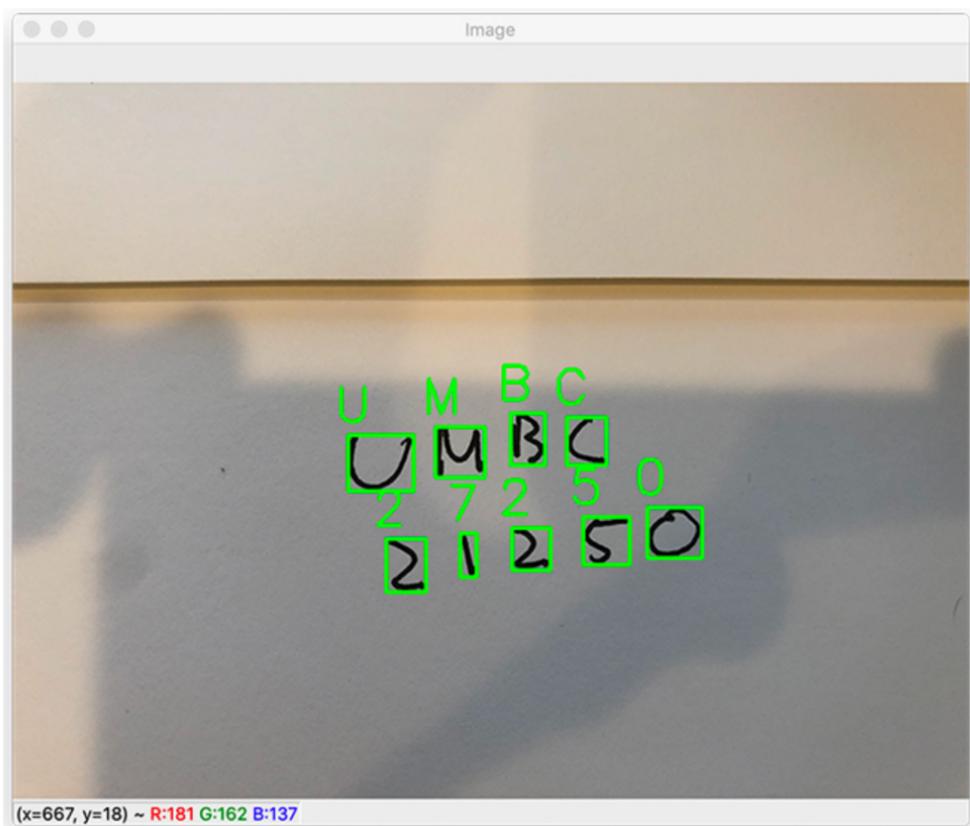
**Figure 3.4.** True to form for programmers, we start with a “HELLO WORLD” example to examine our deep learning OCR model results. But as you can see, the results aren’t quite perfect!

As Figure 3.5 shows, our handwriting recognition algorithm performed *almost perfectly* here. We can correctly OCR every handwritten character in the “UMBC.” However, the ZIP code is incorrectly OCR’d — our model confuses the “1” digit with a “7.” If we were to apply de-skewing to our character data, we could improve our results.

Let’s inspect one final example. This image contains the full address of UMBC:

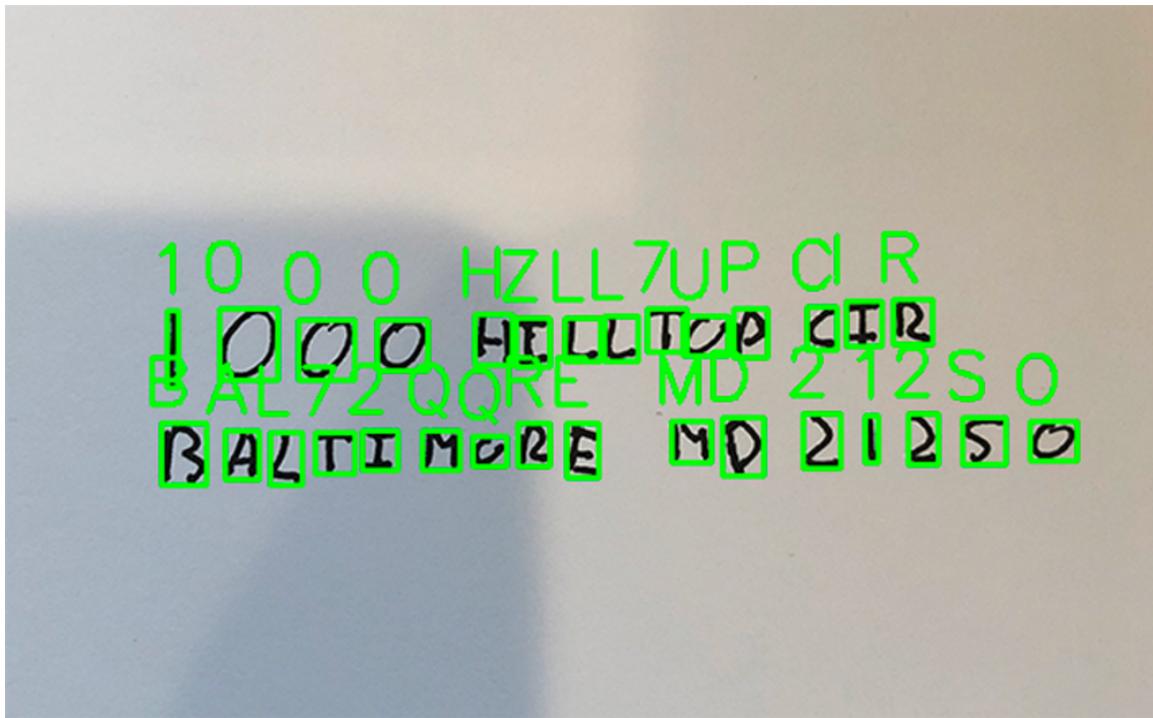
```
$ python ocr_handwriting.py --model handwriting.model --image
→ images/umbc_address.png
[INFO] loading handwriting OCR model...
[INFO] B - 97.71%
[INFO] 1 - 95.41%
[INFO] 0 - 89.55%
[INFO] A - 87.94%
[INFO] L - 96.30%
[INFO] O - 71.02%
[INFO] 7 - 42.04%
[INFO] 2 - 27.84%
[INFO] 0 - 67.76%
[INFO] Q - 28.67%
```

```
[INFO] Q - 39.30%
[INFO] H - 86.53%
[INFO] Z - 61.18%
[INFO] R - 87.26%
[INFO] L - 91.07%
[INFO] E - 98.18%
[INFO] L - 84.20%
[INFO] 7 - 74.81%
[INFO] M - 74.32%
[INFO] U - 68.94%
[INFO] D - 92.87%
[INFO] P - 57.57%
[INFO] 2 - 99.66%
[INFO] C - 35.15%
[INFO] I - 67.39%
[INFO] 1 - 90.56%
[INFO] R - 65.40%
[INFO] 2 - 99.60%
[INFO] S - 42.27%
[INFO] O - 43.73%
```



**Figure 3.5.** For this example, we use an envelope with a name and ZIP code. Not bad — nearly perfect results. The digit “1” was confused for the digit “7.” If we were to apply some computer vision pre-processing, we might be able to improve our results.

Figure 3.6 shows that our handwriting recognition model **struggled**. As you can see, there are multiple mistakes in the words “*Hilltop*,” “*Baltimore*,” and the ZIP code.



**Figure 3.6.** A sample two-line address written on an envelope. We can see there are still multiple mistakes, and thus there are limitations to our OCR model.

Given that our handwriting recognition model performed so well during training and testing, shouldn't we expect it to perform well on our custom images as well?

To answer that question, let's move on to the next section.

### 3.2.5 Limitations, Drawbacks, and Next Steps

While our handwriting recognition model obtained **96% accuracy on our testing set**, our handwriting recognition accuracy on our custom images is slightly less than that.

One of the most significant issues is that we used variants of the MNIST (digits) and NIST (alphabet characters) datasets to train our handwriting recognition model.

While interesting to study, these datasets don't necessarily translate to real-world projects because the images have already been pre-processed and cleaned for us — ***real-world characters aren't that "clean."***

Additionally, our handwriting recognition method requires characters to be individually segmented. That may be possible for *some* characters, but many of us (especially cursive

writers) connect characters when writing quickly. This confuses our model into thinking a *group of characters* is a *single character*, which ultimately leads to incorrect results.

Finally, our model architecture is a bit too simplistic. While our handwriting recognition model performed well on the training and testing set, the architecture — combined with the training dataset itself — is not robust enough to generalize as an “off-the-shelf” handwriting recognition model.

To improve our handwriting recognition accuracy, we should:

- Improve our image pre-processing techniques
- Attempt to remove slant and slope from the handwritten words/characters
- Utilize connectionist temporal classification (CTC) loss
- Implement a novel convolutional recurrent neural network (CRNN) architecture that *combines* convolutional layers with recurrent layers, ultimately yielding higher handwriting recognition accuracy.

### 3.3 Summary

In this chapter, you learned how to perform OCR handwriting recognition using Keras, TensorFlow, and OpenCV.

Our handwriting recognition system utilized basic computer vision and image processing algorithms (edge detection, contours, and contour filtering) to segment characters from an input image.

From there, we passed each character through our trained handwriting recognition model to recognize each character.

Our handwriting recognition model performed well, but there were some cases where the results could have been improved (ideally, with more training data that is *representative* of the handwriting we want to recognize) — **the higher quality the training data, the more accurate we can make our handwriting recognition model!**

Secondly, our handwriting recognition pipeline did not handle the case where characters may be *connected*, causing multiple connected characters to be treated as a *single character*, thus confusing our OCR model.

Dealing with connected handwritten characters is still an open area of research in the computer vision and OCR field; however, deep learning models, specifically LSTMs and recurrent layers, have shown *significant promise* in improving handwriting recognition accuracy.

## Chapter 4

# Using Machine Learning to Denoise Images for Better OCR Accuracy

One of the most challenging aspects of applying optical character recognition (OCR) isn't the *OCR itself*. Instead, it's the process of pre-processing, denoising, and cleaning up images such that they *can* be OCR'd.

When working with documents generated by a computer, screenshots, or essentially any piece of text that has never touched a printer and then scanned, OCR becomes far easier. The text is clean and crisp. There is sufficient contrast between the background and foreground. And most of the time, the text doesn't exist on a complex background.

**That all changes once a piece of text is printed and scanned.** From there, OCR becomes *much* more challenging.

- The printer could be low on toner or ink, resulting in the text appearing faded and hard to read.
- An old scanner could have been used when scanning the document, resulting in low image resolution and poor text contrast.
- A mobile phone scanner app may have been used under poor lighting conditions, making it incredibly challenging for human eyes to read the text, let alone a computer.
- And all too common are the clear signs that an actual human has handled the paper, including coffee mug stains on the corners, paper crinkling, rips, and tears, etc.

For all the amazing things the human mind is capable of, it seems like we're all just walking accidents waiting to happen when it comes to printed materials. Give us a piece of paper and enough time, and I guarantee that even the most organized of us will take that document from the pristine condition and eventually introduce some stains, rips, folds, and crinkles on it.

Inevitably, these problems will occur — and when they do, we need to utilize our computer vision, image processing, and OCR skills such that we can pre-process and improve the quality of these damaged documents. From there, we'll be able to obtain higher OCR accuracy.

In the remainder of this chapter, you'll learn how even simple machine learning algorithms constructed in a novel way can help you denoise images before applying OCR.

## 4.1 Chapter Learning Objectives

In this chapter, you will:

- Gain experience working with a dataset of noisy, damaged documents
- Discover how machine learning can be used to denoise these damaged documents
- Work with Kaggle's Denoising Dirty Documents dataset [9]
- Extract features from this dataset
- Train a random forest regressor (RFR) on the features we extracted
- Take the model and use it to denoise images in our test set (and then be able to denoise your datasets as well)

## 4.2 Image Denoising with Machine Learning

In the first part of this chapter, we will review the dataset we will be using to denoise documents. From there, we'll review our project structure, including the five separate Python scripts we'll be utilizing, including:

- A configuration file to store variables utilized across multiple Python scripts
- A helper function used to blur and threshold our documents
- A script used to extract features and target values from our dataset
- Another script used to train a RFR
- And a final script used to apply our trained model to images in our test set

This is one of the longer chapters in the text, and while it's straightforward and follows a linear progression, there are also many nuanced details here. I suggest you review this chapter *twice*, once at a high-level to understand what we're doing, and then again at a low-level to understand the implementation.

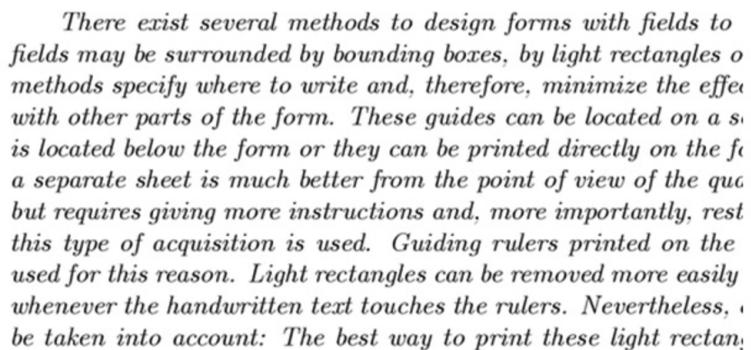
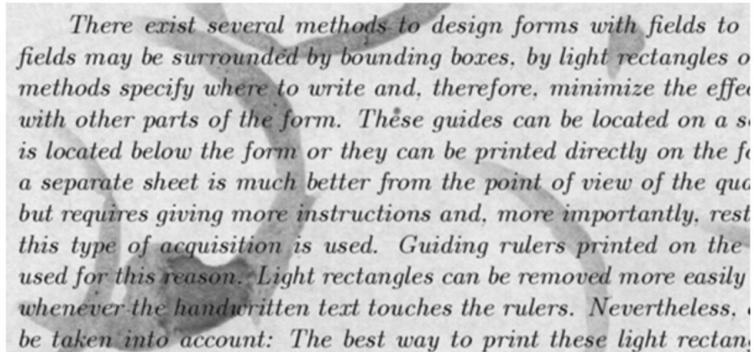
With that said, let's get started!

### 4.2.1 Our Noisy Document Dataset

We'll be using Kaggle's Denoising Dirty Documents dataset (<http://pyimg.co/9cpd6> [9]) in this chapter. The dataset is part of the UCI Machine Learning Repository [10] but converted to a Kaggle competition. We will be using three files for this chapter. Those files are a part of the Kaggle competition data and are named: `test.zip`, `train.zip`, and `train_cleaned.zip`.

The dataset is relatively small, with only 144 training samples, making it easy to work with and use as an educational tool. However, don't let the small dataset size fool you! What we're going to do with this dataset is *far* from basic or introductory.

Figure 4.1 shows a sample of the dirty documents dataset. For the sample document, the *top* shows the document's noisy version, including stains, crinkles, folds, etc. The *bottom* then shows the target, pristine version of the document that we wish to generate.



*There exist several methods to design forms with fields to fields may be surrounded by bounding boxes, by light rectangles o methods specify where to write and, therefore, minimize the effe with other parts of the form. These guides can be located on a s is located below the form or they can be printed directly on the fe a separate sheet is much better from the point of view of the qua but requires giving more instructions and, more importantly, rest this type of acquisition is used. Guiding rulers printed on the used for this reason. Light rectangles can be removed more easily whenever the handwritten text touches the rulers. Nevertheless, be taken into account: The best way to print these light rectan*

**Figure 4.1.** Top: A sample image of a noisy document. Bottom: Target, cleaned version. Our goal is to create a computer vision pipeline that can automatically transform the noisy document into a cleaned one. Image credit: <http://pyimg.co/j6ylr> [11].

**Our goal is to input the image on the *top* and train a machine learning model capable of producing a cleaned output on the *bottom*.** It may seem impossible now, but once you see some of the tricks and techniques we'll be using, it will be a lot more straightforward than you think.

### 4.2.2 The Denoising Document Algorithm

Our denoising algorithm hinges on training an RFR to accept a noisy image and *automatically* predict the output pixel values. This algorithm is inspired by a denoising technique introduced by Colin Priest (<http://pyimg.co/dp06u> [12]).

These algorithms work by applying a  $5 \times 5$  window that slides from *left-to-right* and *top-to-bottom*, one pixel at a time (Figure 4.2) across **both** the noisy image (i.e., the image we want to pre-process automatically and cleanup) and the target output image (i.e., the “gold standard” of what the image should look like after cleaning).

At each sliding window stop, we extract:

- i. The  $5 \times 5$  region of the **noisy, input image**. We then flatten the  $5 \times 5$  region into a  $25-d$  list and treat it like a feature vector.
- ii. The same  $5 \times 5$  region of the **cleaned image**, but this time we only take the center  $(x, y)$ -coordinate, denoted by the location  $(2, 2)$ .

Given the  $25-d$  (dimensional) feature vector from the noisy input image, this single pixel value is what we want our RFR to predict.

To make this example more concrete, again consider Figure 4.2 where we have the following  $5 \times 5$  grid of pixel values from the noisy image:

---

```
[[247 227 242 253 237]
 [244 228 225 212 219]
 [223 218 252 222 221]
 [242 244 228 240 230]
 [217 233 237 243 252]]
```

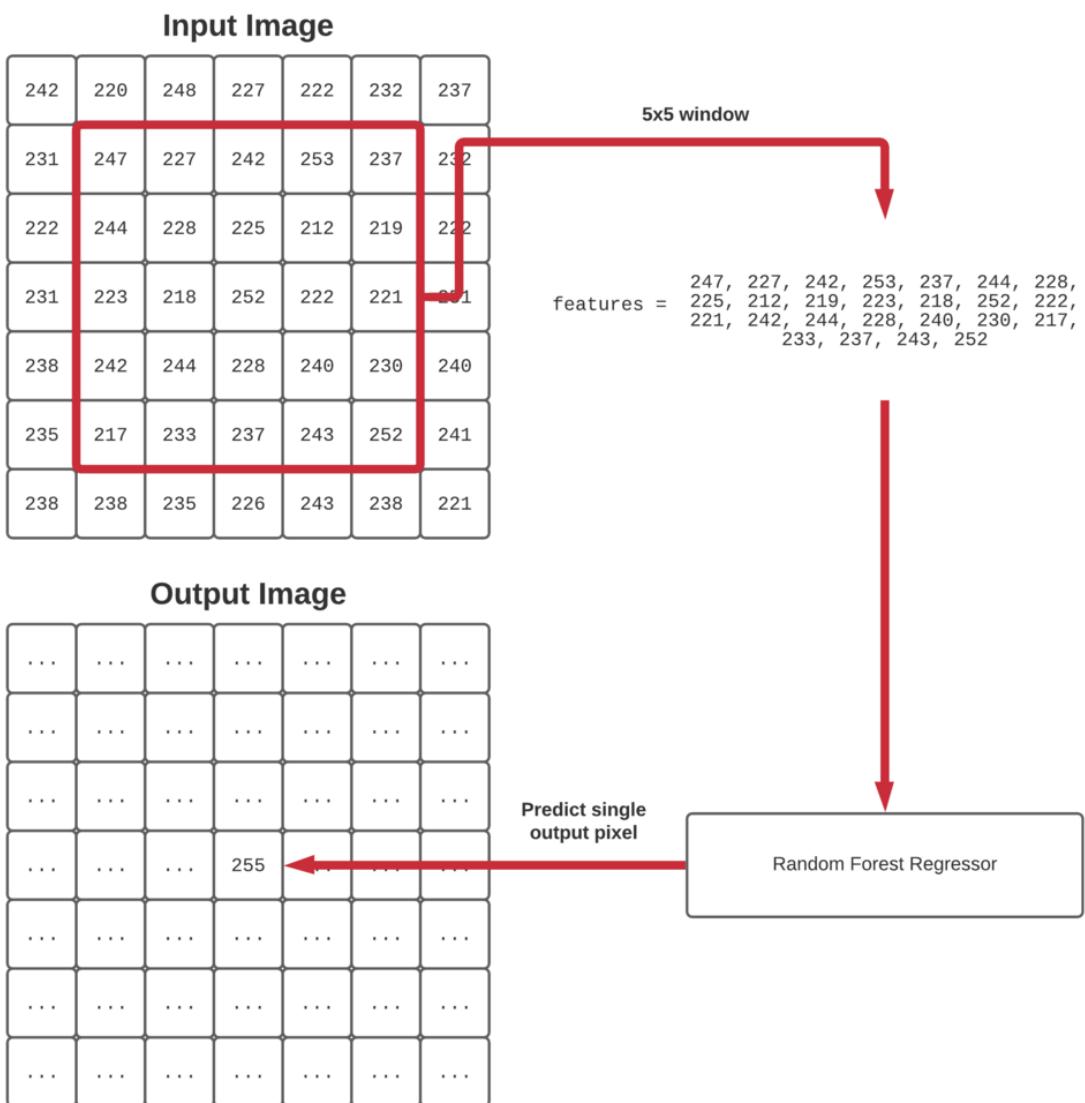
---

We then flatten that into a single list of  $5 \times 5 = 25-d$  values:

---

```
[247 227 242 253 237 244 228 225 212 219 223 218 252 222 221 242 244 228
 240 230 217 233 237 243 252]
```

---



**Figure 4.2.** Our denoising document algorithm works by sliding a  $5 \times 5$  window from *left-to-right* and *top-to-bottom* across the noisy input. We extract this  $5 \times 5$  region, flatten it into a 25-d vector, and then treat it as our feature vector. This feature vector is passed into our RFR, and the output, a cleaned pixel, is predicted.

**This 25-d vector is our feature vector upon which our RFR will be trained.**

However, we still need to define the target output value of the RFR. **Our regression model should accept the input 25-d vector and output the cleaned, denoised pixel.**

Now, let's assume that we have the following  $5 \times 5$  window from our gold standard/target image:

---

```
[[0 0 0 0 0]
 [0 0 0 0 1]
 [0 0 1 1 1]
 [0 0 1 1 1]
 [0 0 0 1 1]]
```

---

We are only interested in the *center* of this  $5 \times 5$  *region*, denoted as the location  $x = 2, y = 2$ . **We extract this value of 1 (foreground, versus 0, which is background) and treat it as our target value that our RFR should predict.**

Putting this entire example together, we can think of the following as a sample training data point:

---

```
1 trainX = [[247 227 242 253 237 244 228 225 212 219 223 218 252 222 221 242 244 228
2 240 230 217 233 237 243 252]]
3 trainY = [[1]]
```

---

Given our `trainX` variable (which is our raw pixel intensities), we want to be able to predict the corresponding cleaned/denoised pixel value in `trainY`.

**We will train our RFR in this manner, ultimately leading to a model that can accept a noisy document input and automatically denoise it by examining local  $5 \times 5$  regions and then predicting the center (cleaned) pixel value.**

### 4.2.3 Project Structure

This chapter's project directory structure is a bit more complex than previous chapters as there are five Python scripts to review (three scripts, a helper function, and a configuration file).

Before we get any farther, let's familiarize ourselves with the files:

---

```
|-- pyimagesearch
|   |-- __init__.py
|   |-- denoising
|       |-- __init__.py
|       |-- helpers.py
|-- config
|   |-- __init__.py
|   |-- denoise_config.py
|-- build_features.py
```

---

```
|-- denoise_document.py
|-- denoiser.pickle
|-- denoising-dirty-documents
|   |-- test
|   |   |-- 1.png
|   |   |-- 10.png
|   |   |-- ...
|   |   |-- 94.png
|   |   |-- 97.png
|   |-- train
|   |   |-- 101.png
|   |   |-- 102.png
|   |   |-- ...
|   |   |-- 98.png
|   |   |-- 99.png
|   |-- train_cleaned
|   |   |-- 101.png
|   |   |-- 102.png
|   |   |-- ...
|   |   |-- 98.png
|   |   |-- 99.png
|-- train_denoiser.py
```

---

The `denoising-dirty-documents` directory contains all images from the Kaggle Denoising Dirty Documents dataset [9].

Inside the `denoising` submodule of `pyimagesearch`, there is a `helpers.py` file. This file contains a single function, `blur_and_threshold`, which, as the name suggests, is used to apply a combination of smoothing and thresholding as a pre-processing step for our documents.

We then have the `denoise_config.py` file, which stores a small number of configurations, namely specifying training data file paths, output feature CSV files, and the final serialized RFR model.

There are three Python scripts that we'll be reviewing in entirety:

- i. `build_features.py`: Accepts our input dataset and creates a CSV file that we'll use to train our RFR.
- ii. `train_denoiser.py`: Trains the actual RFR model and serializes it to disk as `denoiser.pickle`.
- iii. `denoise_document.py`: Accepts an input image from disk, loads the trained RFR, and then denoises the input image.

There are several Python scripts that we need to review in this chapter. I suggest you review this chapter *twice* to gain a higher-level understanding of what we are implementing and then again to grasp the implementation at a deeper level.

#### 4.2.4 Implementing Our Configuration File

The first step in our denoising documents implementation is to create our configuration file.

Open the `denoise_config.py` file in the `config` subdirectory of the project directory structure and insert the following code:

---

```

1 # import the necessary packages
2 import os
3
4 # initialize the base path to the input documents dataset
5 BASE_PATH = "denoising-dirty-documents"
6
7 # define the path to the training directories
8 TRAIN_PATH = os.path.sep.join([BASE_PATH, "train"])
9 CLEANED_PATH = os.path.sep.join([BASE_PATH, "train_cleaned"])

```

---

**Line 5** defines the base path to our `denoising-dirty-documents` dataset. If you download this dataset from Kaggle, be sure to unzip *all* `.zip` files within this directory to have all images in the dataset uncompressed and residing on disk.

We then define the paths to both the *original, noisy* image directory and the corresponding *cleaned image* directory, respectively (**Lines 8 and 9**).

The `TRAIN_PATH` images contain the noisy documents while the `CLEANED_PATH` images contain our “gold standard” of what, ideally, our output images should look like after applying document denoising via our trained model. We’ll construct our testing set inside our `train_denoiser.py` script.

Let’s continue defining our configuration file:

---

```

11 # define the path to our output features CSV file then initialize
12 # the sampling probability for a given row
13 FEATURES_PATH = "features.csv"
14 SAMPLE_PROB = 0.02
15
16 # define the path to our document denoiser model
17 MODEL_PATH = "denoiser.pickle"

```

---

**Line 13** defines the path to our output `features.csv` file. Our features here will consist of:

- i. A local  $5 \times 5$  region sampled via sliding window from the noisy input image
- ii. The center of the  $5 \times 5$  region, denoted as  $(x, y)$ -coordinate  $(2, 2)$ , for the corresponding cleaned image

However, if we wrote *every* feature/target combination to disk, we would end up with *millions* of rows and a CSV many gigabytes in size. Instead of exhaustively computing all sliding window and target combinations, we'll instead only write them to disk with SAMPLES\_PROB probability.

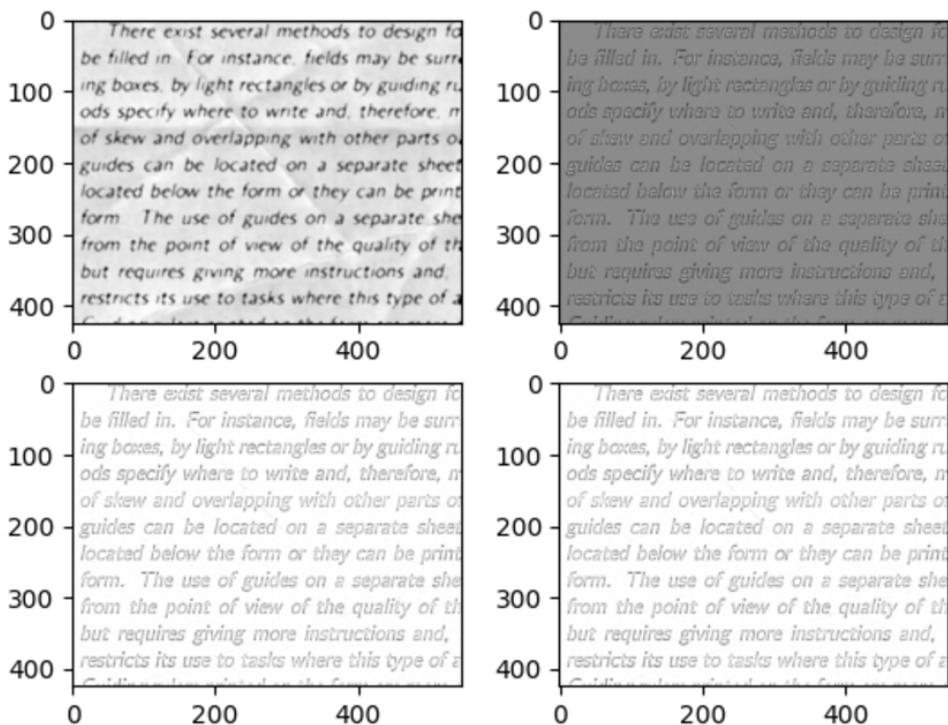
Finally, **Line 17** specifies the path to MODEL\_PATH, our output serialized model.

#### 4.2.5 Creating Our Blur and Threshold Helper Function

To help our RFR predict background (i.e., noisy) from foreground (i.e., text) pixels, we need to define a helper function that will pre-process our images before we train the model and make predictions with it.

The flow of our image processing operations can be seen in Figure 4.3. First, we take our input image, blur it (*top-left*), and then subtract the blurred image from the input image (*top-right*). **We do this step to approximate the foreground of the image since, by nature, blurring will blur focused features and reveal more of the “structural” components of the image.**

**Blur and Threshold Preprocessing**



**Figure 4.3.** *Top-left:* Applying a small median blur to the input image. *Top-right:* Subtracting the blurred image from the original image. *Bottom-left:* Thresholding the subtracted image. *Bottom-right:* Performing min-max scaling to ensure the pixel intensities are in the range  $[0, 1]$ .

Next, we threshold the approximate foreground region by setting any pixel values greater than zero to zero (Figure 4.3, *bottom-left*).

The final step is to perform min-max scaling (*bottom-right*), which brings the pixel intensities back to the range  $[0, 1]$  (or  $[0, 255]$ , depending on your data type). **This final image will serve as noisy input when we perform our sliding window sampling.**

Now that we understand the general pre-processing steps, let's implement them in Python code.

Open the `helpers.py` file in the denoising submodule of `pyimagesearch` and let's get to work defining our `blur_and_threshold` function:

---

```

1 # import the necessary packages
2 import numpy as np
3 import cv2
4
5 def blur_and_threshold(image, eps=1e-7):
6     # apply a median blur to the image and then subtract the blurred
7     # image from the original image to approximate the foreground
8     blur = cv2.medianBlur(image, 5)
9     foreground = image.astype("float") - blur
10
11    # threshold the foreground image by setting any pixels with a
12    # value greater than zero to zero
13    foreground[foreground > 0] = 0

```

---

The `blur_and_threshold` function accepts two parameters:

- i. `image`: Our input image that we'll be pre-processing.
- ii. `eps`: An epsilon value used to prevent division by zero.

We then apply a median blur to the image to reduce noise and then subtract the `blur` from the original `image`, resulting in a foreground approximation (**Lines 8 and 9**).

From there, we threshold the `foreground` image by setting any pixel intensities greater than zero to zero (**Line 13**).

The final step here is to perform min-max scaling:

---

```

15    # apply min/max scaling to bring the pixel intensities to the
16    # range [0, 1]
17    minValue = np.min(foreground)
18    maxValue = np.max(foreground)
19    foreground = (foreground - minValue) / (maxValue - minValue + eps)
20

```

---

---

```
21     # return the foreground-approximated image
22     return foreground
```

---

Here we find the minimum and maximum values in the `foreground` image. We use these values to scale the pixel intensities in the `foreground` image to the range  $[0, 1]$ .

This foreground-approximated image is then returned to the calling function.

#### 4.2.6 Implementing the Feature Extraction Script

With our `blur_and_threshold` function defined, we can move on to our `build_features.py` script.

As the name suggests, this script is responsible for creating our  $5 \times 5 - 25-d$  feature vectors from the *noisy* image and then extracting the *target* (i.e., cleaned) pixel value from the corresponding gold standard image.

We'll save these features to disk in CSV format and then train a Random Forest Regression model on them in Section 4.2.8.

Let's get started with our implementation now:

---

```
1 # import the necessary packages
2 from config import denoise_config as config
3 from pyimagesearch.denoising import blur_and_threshold
4 from imutils import paths
5 import progressbar
6 import random
7 import cv2
```

---

**Line 2** imports our `config` to have access to our dataset file paths and output CSV file path. Notice that we're using the `blur_and_threshold` function that we implemented in the previous section.

The following code block grabs the paths to all images in our `TRAIN_PATH` (noisy images) and `CLEANED_PATH` (cleaned images that our RFR will learn to predict):

---

```
9 # grab the paths to our training images
10 trainPaths = sorted(list(paths.list_images(config.TRAIN_PATH)))
11 cleanedPaths = sorted(list(paths.list_images(config.CLEANED_PATH)))
12
13 # initialize the progress bar
14 widgets = ["Creating Features: ", progressbar.Percentage(), " ",
15             progressbar.Bar(), " ", progressbar.ETA()]
```

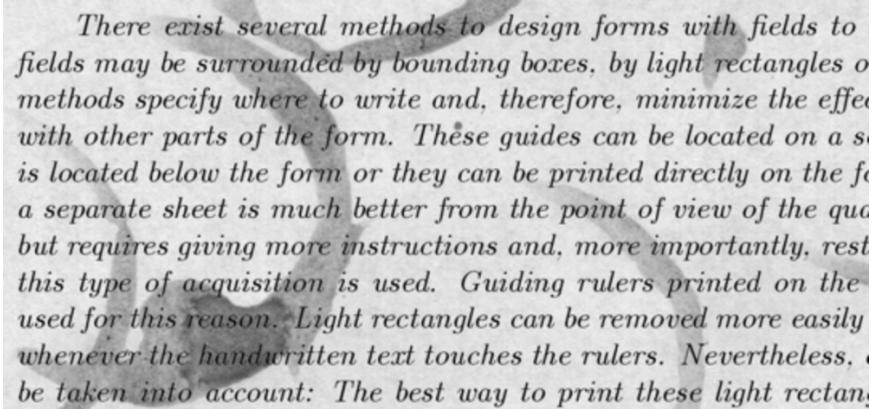
---

```
16 pbar = progressbar.ProgressBar(maxval=len(trainPaths),
17     widgets=widgets).start()
```

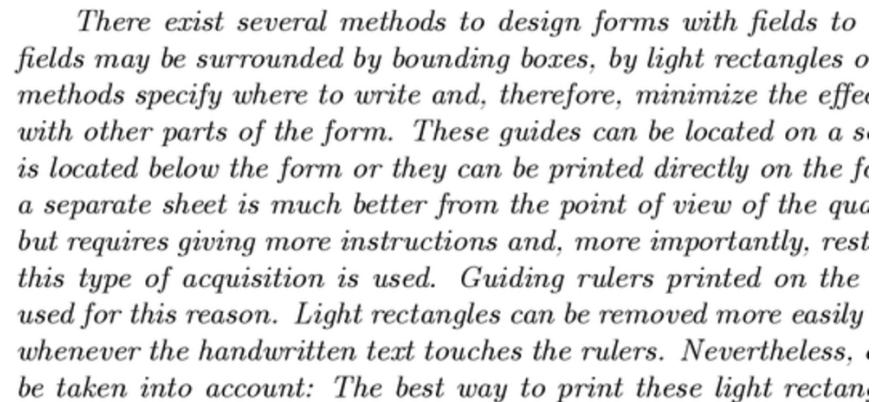
---

Note that `trainPaths` contain all our *noisy* images. The `cleanedPaths` contain the corresponding *cleaned* images.

Figure 4.4 shows an example. On the *top* is our input training image. On the *bottom*, we have the corresponding cleaned version of the image. We'll take  $5 \times 5$  regions from both the `trainPaths` and the `cleanedPaths` — the goal is to use the noisy  $5 \times 5$  regions to predict the cleaned versions.



*There exist several methods to design forms with fields to be filled in. Fields may be surrounded by bounding boxes, by light rectangles or other methods. These methods specify where to write and, therefore, minimize the effect of the handwritten text on the rest of the form. These guides can be located on a separate sheet or directly on the form. A separate sheet is much better from the point of view of the quality of the handwriting, but requires giving more instructions and, more importantly, rests on the form. This type of acquisition is used. Guiding rulers printed on the form are used for this reason. Light rectangles can be removed more easily whenever the handwritten text touches the rulers. Nevertheless, some care must be taken into account: The best way to print these light rectangles is to use a separate sheet.*



*There exist several methods to design forms with fields to be filled in. Fields may be surrounded by bounding boxes, by light rectangles or other methods. These methods specify where to write and, therefore, minimize the effect of the handwritten text on the rest of the form. These guides can be located on a separate sheet or directly on the form. A separate sheet is much better from the point of view of the quality of the handwriting, but requires giving more instructions and, more importantly, rests on the form. This type of acquisition is used. Guiding rulers printed on the form are used for this reason. Light rectangles can be removed more easily whenever the handwritten text touches the rulers. Nevertheless, some care must be taken into account: The best way to print these light rectangles is to use a separate sheet.*

**Figure 4.4.** *Top:* A sample training image of a noisy input document. *Bottom:* The corresponding target/gold standard for the training image. Our goal is to train a model such that we can take the *top* input and automatically generate the *bottom* output.

Let's start looping over these image combinations now:

---

```
19 # zip our training paths together, then open the output CSV file for
20 # writing
21 imagePaths = zip(trainPaths, cleanedPaths)
22 csv = open(config.FEATURES_PATH, "w")
23
24 # loop over the training images together
```

---

---

```

25  for (i, (trainPath, cleanedPath)) in enumerate(imagePaths):
26      # load the noisy and corresponding gold-standard cleaned images
27      # and convert them to grayscale
28      trainImage = cv2.imread(trainPath)
29      cleanImage = cv2.imread(cleanedPath)
30      trainImage = cv2.cvtColor(trainImage, cv2.COLOR_BGR2GRAY)
31      cleanImage = cv2.cvtColor(cleanImage, cv2.COLOR_BGR2GRAY)

```

---

On **Line 21** we use Python's `zip` function to combine the `trainPaths` and `cleanedPaths` together. We then open our output `csv` file for writing on **Line 22**.

**Line 25** starts a loop over our combinations of `imagePaths`. For each `trainPath`, we also have the corresponding `cleanedPath`.

We load our `trainImage` and `cleanImage` from disk and then convert them to grayscale (**Lines 28–31**).

Next, we need to pad both `trainImage` and `cleanImage` with a 2-pixel border in every direction:

---

```

33      # apply 2x2 padding to both images, replicating the pixels along
34      # the border/boundary
35      trainImage = cv2.copyMakeBorder(trainImage, 2, 2, 2, 2,
36          cv2.BORDER_REPLICATE)
37      cleanImage = cv2.copyMakeBorder(cleanImage, 2, 2, 2, 2,
38          cv2.BORDER_REPLICATE)
39
40      # blur and threshold the noisy image
41      trainImage = blur_and_threshold(trainImage)
42
43      # scale the pixel intensities in the cleaned image from the range
44      # [0, 255] to [0, 1] (the noisy image is already in the range
45      # [0, 1])
46      cleanImage = cleanImage.astype("float") / 255.0

```

---

Why do we need to bother with the padding? We're sliding a window from *left-to-right* and *top-to-bottom* of the input image and using the pixels inside the window to predict the output *center* pixel located at  $x = 2, y = 2$ , not unlike a convolution operation (only with convolution our filters are fixed and defined).

Like convolution, you need to pad your input images such that the output image is not smaller in size. You can refer to my guide on *Convolutions with OpenCV and Python* if you are unfamiliar with the concept: <http://pyimg.co/8om5g> [13].

After padding is complete, we blur and threshold the `trainImage` and manually scale the `cleanImage` to the range  $[0, 1]$ . The `trainImage` is *already* scaled to the range  $[0, 1]$  due to the min-max scaling inside `blur_and_threshold`.

With our images pre-processed, we can now slide a  $5 \times 5$  window across them:

---

```

48     # slide a 5x5 window across the images
49     for y in range(0, trainImage.shape[0]):
50         for x in range(0, trainImage.shape[1]):
51             # extract the window ROIs for both the train image and
52             # clean image, then grab the spatial dimensions of the
53             # ROI
54             trainROI = trainImage[y:y + 5, x:x + 5]
55             cleanROI = cleanImage[y:y + 5, x:x + 5]
56             (rH, rW) = trainROI.shape[:2]
57
58             # if the ROI is not 5x5, throw it out
59             if rW != 5 or rH != 5:
60                 continue

```

---

**Lines 49 and 50** slide a  $5 \times 5$  window from left-to-right and top-to-bottom across the `trainImage` and `cleanImage`. At each sliding window stop, we extract the  $5 \times 5$  ROI of the training image and clean image (**Lines 54 and 55**).

We grab the width and height of the `trainROI` on **Line 56**, and if either the width or height is not five pixels (due to us being on the borders of the image), we throw out the ROI (because we are only concerned with  $5 \times 5$  regions).

Next, we construct our feature vectors and save the row to our CSV file:

---

```

62             # our features will be the flattened 5x5=25 raw pixels
63             # from the noisy ROI while the target prediction will
64             # be the center pixel in the 5x5 window
65             features = trainROI.flatten()
66             target = cleanROI[2, 2]
67
68             # if we wrote *every* feature/target combination to disk
69             # we would end up with millions of rows -- let's only
70             # write rows to disk with probability N, thereby reducing
71             # the total number of rows in the file
72             if random.random() <= config.SAMPLE_PROB:
73                 # write the target and features to our CSV file
74                 features = [str(x) for x in features]
75                 row = [str(target)] + features
76                 row = ",".join(row)
77                 csv.write("{}\n".format(row))
78
79             # update the progress bar
80             pbar.update(i)
81
82     # close the CSV file
83     pbar.finish()
84     csv.close()

```

---

**Line 65** takes the  $5 \times 5$  pixel region from the `trainROI` and flattens it into a  $5 \times 5 = 25$ -d list — **this list serves as our feature vector**.

**Line 66** then extracts the *cleaned/gold-standard* pixel value from the center of the `cleanROI`. **This pixel value serves as what we want our RFR to predict.**

At this point, we could write our combination of a feature vector and target value to disk; however, if we were to write *every* feature/target combination to the CSV file, we would end up with a file many gigabytes in size.

To avoid having a massive CSV file, we would need to process in the next step, we instead only allow `SAMPLE_PROB` (in this case, 2%) of the rows to be written to disk (**Line 72**). Doing this sampling reduces the resulting CSV file size and makes it easier to manage.

**Line 74** constructs our row of `features` and prepends the target pixel value to it. We then write the row to our CSV file. We repeat this process for all `imagePaths`.

#### 4.2.7 Running the Feature Extraction Script

We are now ready to run our feature extractor. Open a terminal and then execute the `build_features.py` script:

---

```
$ python build_features.py
Creating Features: 100% |#####| Time: 0:01:05
```

---

On my 3 GHz Intel Xeon W processor, the entire feature extractor process took just over one minute.

Inspecting my project directory structure, you can now see the resulting CSV file of features:

---

```
$ ls -l *.csv
adrianrosebrock  staff  273968497 Oct 23 06:21 features.csv
```

---

If you were to open the `features.csv` file in your system, you would see that each row contains 26 entries.

The first entry in the row is the target output pixel. We will try to predict the output pixel value based on the contents of the remainder of the row, which are the  $5 \times 5 = 25$  input ROI pixels.

The next section covers how to train an RFR model to do exactly that.

### 4.2.8 Implementing Our Denoising Training Script

Now that our `features.csv` file has been generated, we can move on to the training script. This script is responsible for loading our `features.csv` file and training an RFR to accept a  $5 \times 5$  region of a *noisy* image and then predict the *cleaned* center pixel value.

Let's get started reviewing the code:

---

```

1 # import the necessary packages
2 from config import denoise_config as config
3 from sklearn.ensemble import RandomForestRegressor
4 from sklearn.metrics import mean_squared_error
5 from sklearn.model_selection import train_test_split
6 import numpy as np
7 import pickle

```

---

**Lines 2–7** handle our required Python packages, including:

- `config`: Our project configuration holding our output file paths and training variables
- `RandomForestRegressor`: The scikit-learn implementation of the regression model we'll use to predict pixel values
- `mean_squared_error`: Our error/loss function — the lower this value, the better job we are doing at denoising our images
- `train_test_split`: Used to create a training/testing split from our `features.csv` file
- `pickle`: Used to serialize our trained RFR to disk

Let's move on to loading our CSV file from disk:

---

```

9 # initialize lists to hold our features and target predicted values
10 print("[INFO] loading dataset...")
11 features = []
12 targets = []
13
14 # loop over the rows in our features CSV file
15 for row in open(config.FEATURES_PATH):
16     # parse the row and extract (1) the target pixel value to predict
17     # along with (2) the 5x5=25 pixels which will serve as our feature
18     # vector
19     row = row.strip().split(", ")
20     row = [float(x) for x in row]
21     target = row[0]
22     pixels = row[1:]

```

---

---

```

23
24     # update our features and targets lists, respectively
25     features.append(pixels)
26     targets.append(target)

```

---

**Lines 11 and 12** initialize our `features` ( $5 \times 5$  pixel regions) and `targets` (target output pixel values we want to predict).

We start looping over all lines of our CSV file on **Line 15**. For each `row`, we extract both the target and pixel values (**Lines 19–22**). We then update both our `features` and `targets` lists, respectively.

With the CSV file loaded into memory, we can construct our training and testing split:

---

```

28 # convert the features and targets to NumPy arrays
29 features = np.array(features, dtype="float")
30 target = np.array(targets, dtype="float")
31
32 # construct our training and testing split, using 75% of the data for
33 # training and the remaining 25% for testing
34 (trainX, testX, trainY, testY) = train_test_split(features, target,
35     test_size=0.25, random_state=42)

```

---

Here we use 75% of our data for training and mark the remaining 25% for testing. This type of split is fairly standard in the machine learning field.

Finally, we can train our RFR:

---

```

37 # train a random forest regressor on our data
38 print("[INFO] training model...")
39 model = RandomForestRegressor(n_estimators=10)
40 model.fit(trainX, trainY)
41
42 # compute the root mean squared error on the testing set
43 print("[INFO] evaluating model...")
44 preds = model.predict(testX)
45 rmse = np.sqrt(mean_squared_error(testY, preds))
46 print("[INFO] rmse: {}".format(rmse))
47
48 # serialize our random forest regressor to disk
49 f = open(config.MODEL_PATH, "wb")
50 f.write(pickle.dumps(model))
51 f.close()

```

---

**Line 39** initializes our `RandomForestRegressor`, instructing it to train 10 separate regression trees. The model is then trained on **Line 40**.

After training is complete, we compute the root-mean-square error (RMSE) to measure how good a job we've done at predicting cleaned, denoised images. **The lower the error value, the better the job we've done.**

Finally, we serialize our trained RFR model to disk such that we can use it to make predictions on our noisy images in Section 4.2.10.

#### 4.2.9 Training Our Document Denoising Model

With our `train_denoiser.py` script implemented, we are now ready to train our automatic image denoiser! Open a shell and then execute the `train_denoiser.py` script:

---

```
$ time python train_denoiser.py
[INFO] loading dataset...
[INFO] training model...
[INFO] evaluating model...
[INFO] rmse: 0.04990744293857625

real    1m18.708s
user    1m19.361s
sys     0m0.894s
```

---

Training our script takes just over one minute, resulting in an RMSE of  $\approx 0.05$ . This is a very low loss value, indicating that our model has done a good job of accepting noisy input pixel ROIs and correctly predicting the target output value.

Inspecting our project directory structure, you'll see that the RFR model has been serialized to disk as `denoiser.pickle`:

---

```
$ ls -l *.pickle
adrianrosebrock  staff  77733392 Oct 23 denoiser.pickle
```

---

We'll load our trained `denoiser.pickle` model from disk in the next section and then use it to automatically clean and pre-process our input documents.

#### 4.2.10 Creating the Document Denoiser Script

This project's final step is to take our trained denoiser model to clean our input images *automatically*.

Open `denoise_document.py` now, and we'll see how this process is done:

---

```

1 # import the necessary packages
2 from config import denoise_config as config
3 from pyimagesearch.denoising import blur_and_threshold
4 from imutils import paths
5 import argparse
6 import pickle
7 import random
8 import cv2

```

---

Lines 2–8 handle importing our required Python packages. We then move on to parsing our command line arguments:

---

```

10 # construct the argument parse and parse the arguments
11 ap = argparse.ArgumentParser()
12 ap.add_argument("-t", "--testing", required=True,
13     help="path to directory of testing images")
14 ap.add_argument("-s", "--sample", type=int, default=10,
15     help="sample size for testing images")
16 args = vars(ap.parse_args())

```

---

Our `denoise_document.py` script accepts two command line arguments:

- i. `--testing`: The path to the directory containing the testing images for Kaggle's *Denoising Dirty Documents* [9] dataset
- ii. `--sample`: Number of testing images we'll sample when applying our denoising model

Speaking of our denoising model, let's load the serialized model from disk:

---

```

18 # load our document denoiser from disk
19 model = pickle.loads(open(config.MODEL_PATH, "rb").read())
20
21 # grab the paths to all images in the testing directory and then
22 # randomly sample them
23 imagePaths = list(paths.list_images(args["testing"]))
24 random.shuffle(imagePaths)
25 imagePaths = imagePaths[:args["sample"]]

```

---

We also grab all `imagePaths` part of the testing set, randomly shuffle them, and then select a total of `--sample` images to apply our automatic denoiser model to.

Let's loop over our sample of `imagePaths`:

---

```

27 # loop over the sampled image paths
28 for imagePath in imagePaths:

```

---

```

29     # load the image, convert it to grayscale, and clone it
30     print("[INFO] processing {}".format(imagePath))
31     image = cv2.imread(imagePath)
32     image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
33     orig = image.copy()
34
35     # pad the image followed by blurring/thresholding it
36     image = cv2.copyMakeBorder(image, 2, 2, 2, 2,
37                               cv2.BORDER_REPLICATE)
38     image = blur_and_threshold(image)

```

---

Here we are performing the same pre-processing steps that we utilized during the training phase:

- We load the input image from disk
- Convert it to grayscale
- Pad the image with two pixels in every direction
- Apply the `blur_and_threshold` function

Now we need to loop over the processed image and extract **every**  $5 \times 5$  pixel neighborhood:

---

```

40     # initialize a list to store our ROI features (i.e., 5x5 pixel
41     # neighborhoods)
42     roiFeatures = []
43
44     # slide a 5x5 window across the image
45     for y in range(0, image.shape[0]):
46         for x in range(0, image.shape[1]):
47             # extract the window ROI and grab the spatial dimensions
48             roi = image[y:y + 5, x:x + 5]
49             (rH, rW) = roi.shape[:2]
50
51             # if the ROI is not 5x5, throw it out
52             if rW != 5 or rH != 5:
53                 continue
54
55             # our features will be the flattened 5x5=25 pixels from
56             # the training ROI
57             features = roi.flatten()
58             roiFeatures.append(features)

```

---

**Line 42** initializes a list, `roiFeatures`, to store every  $5 \times 5$  neighborhood.

We then slide a  $5 \times 5$  window from *left-to-right* and *top-to-bottom* across the image. At every step of the window, we extract the `roi` (**Line 48**), grab its spatial dimensions (**Line 49**), and throw it out if the ROI size is not  $5 \times 5$  (**Lines 52 and 53**).

We then take our  $5 \times 5$  pixel neighborhood, flatten it into a list of `features`, and update our `roiFeatures` list (**Lines 57 and 58**).

Outside of our sliding window `for` loops now, we have our `roiFeatures` populated with every possible  $5 \times 5$  pixel neighborhood.

We can then make predictions on these `roiFeatures`, resulting in the final cleaned image:

---

```
60     # use the ROI features to predict the pixels of our new denoised
61     # image
62     pixels = model.predict(roiFeatures)
63
64     # the pixels list is currently a 1D array so we need to reshape
65     # it to a 2D array (based on the original input image dimensions)
66     # and then scale the pixels from the range [0, 1] to [0, 255]
67     pixels = pixels.reshape(orig.shape)
68     output = (pixels * 255).astype("uint8")
69
70     # show the original and output images
71     cv2.imshow("Original", orig)
72     cv2.imshow("Output", output)
73     cv2.waitKey(0)
```

---

**Line 62** calls the `.predict` method our RFR, resulting in `pixels`, which are our foreground versus background predictions.

However, our `pixels` list is currently a 1-d array, so we must take care to `reshape` the array into a 2-d image and then scale the pixel intensities back to the range  $[0, 255]$  (**Lines 67 and 68**).

Finally, we can show both the original (noisy image) and the output (cleaned image) to our screen.

#### 4.2.11 Running our Document Denoiser

You made it! This has been a long chapter, but we've finally ready to apply our document denoiser to our test data.

To see our `denoise_document.py` script in action, open a terminal and execute the following command:

---

```
$ python denoise_document.py --testing denoising-dirty-documents/test
[INFO] processing denoising-dirty-documents/test/133.png
[INFO] processing denoising-dirty-documents/test/160.png
[INFO] processing denoising-dirty-documents/test/40.png
[INFO] processing denoising-dirty-documents/test/28.png
```

```
[INFO] processing denoising-dirty-documents/test/157.png
[INFO] processing denoising-dirty-documents/test/190.png
[INFO] processing denoising-dirty-documents/test/100.png
[INFO] processing denoising-dirty-documents/test/49.png
[INFO] processing denoising-dirty-documents/test/58.png
[INFO] processing denoising-dirty-documents/test/10.png
```

---

Our results can be seen in Figure 4.5. The *left* image for each sample shows the noisy input document, including stains, crinkles, folds, etc. The *right* then shows the output cleaned image as generated by our RFR.

A new offline handwritten database for the Spanish language, which contains full Spanish sentences, has recently been developed: the Spartacus database (which stands for Spanish Restricted-domain Task of Cursive Sentences). There were two main reasons for creating this corpus. First of all, most databases do not contain Spanish sentences, even though Spanish is a widespread major language. Another important reason was to create a corpus from semantic tasks. These tasks are commonly used in practice of linguistic knowledge beyond the lexicon level in the recognition process.

As the Spartacus database consisted mainly of short sentences and did not contain long paragraphs, the writers were asked to copy fixed places: dedicated one-line fields in the forms. Next, the forms used in the acquisition process. These forms contained instructions given to the writer.

A new offline handwritten database for the Spanish language, which contains full Spanish sentences, has recently been developed: the Spartacus database (which stands for Spanish Restricted-domain Task of Cursive Sentences). There were two main reasons for creating this corpus. First of all, most databases do not contain Spanish sentences, even though Spanish is a widespread major language. Another important reason was to create a corpus from semantic tasks. These tasks are commonly used in practice of linguistic knowledge beyond the lexicon level in the recognition process.

A new offline handwritten database for the Spanish language, which contains full Spanish sentences, has recently been developed: the Spartacus database (which stands for Spanish Restricted-domain Task of Cursive Sentences). There were two main reasons for creating this corpus. First of all, most databases do not contain Spanish sentences, even though Spanish is a widespread major language. Another important reason was to create a corpus from semantic tasks. These tasks are commonly used in practice and allow the use of linguistic knowledge beyond the lexicon level in the recognition process.

As the Spartacus database consisted mainly of short sentences and did not contain long paragraphs, the writers were asked to copy fixed places: dedicated one-line fields in the forms.

A new offline handwritten database for the Spanish language, which contains full Spanish sentences, has recently been developed: the Spartacus database (which stands for Spanish Restricted-domain Task of Cursive Sentences). There were two main reasons for creating this corpus. First of all, most databases do not contain Spanish sentences, even though Spanish is a widespread major language. Another important reason was to create a corpus from semantic tasks. These tasks are commonly used in practice of linguistic knowledge beyond the lexicon level in the recognition process.

As the Spartacus database consisted mainly of short sentences and did not contain long paragraphs, the writers were asked to copy fixed places: dedicated one-line fields in the forms. Next, the forms used in the acquisition process. These forms contained instructions given to the writer.

A new offline handwritten database for the Spanish language, which contains full Spanish sentences, has recently been developed: the Spartacus database (which stands for Spanish Restricted-domain Task of Cursive Sentences). There were two main reasons for creating this corpus. First of all, most databases do not contain Spanish sentences, even though Spanish is a widespread major language. Another important reason was to create a corpus from semantic tasks. These tasks are commonly used in practice of linguistic knowledge beyond the lexicon level in the recognition process.

A new offline handwritten database for the Spanish language, which contains full Spanish sentences, has recently been developed: the Spartacus database (which stands for Spanish Restricted-domain Task of Cursive Sentences). There were two main reasons for creating this corpus. First of all, most databases do not contain Spanish sentences, even though Spanish is a widespread major language. Another important reason was to create a corpus from semantic tasks. These tasks are commonly used in practice and allow the use of linguistic knowledge beyond the lexicon level in the recognition process.

As the Spartacus database consisted mainly of short sentences and did not contain long paragraphs, the writers were asked to copy fixed places: dedicated one-line fields in the forms.

**Figure 4.5.** A sample of three noisy paragraph inputs from three different printings of the same document. The samples are shown on the (*left*) and their corresponding cleaned outputs are shown on the *right*. Notice how the image quality has *significantly* improved by running our RFR on these images!

**As you can see, our RFR is doing a great job cleaning these images for us automatically!**

## 4.3 Summary

In this chapter, you learned how to denoise dirty documents using computer vision and machine learning.

Using this method, we could accept images of documents that had been “damaged”, including rips, tears, stains, crinkles, folds, etc. By applying machine learning in a novel way, we could clean up these images to near *pristine* conditions, thereby making it easier for OCR engines to detect the text, extract it, and OCR it correctly.

When you find yourself applying OCR to real-world images, *especially* scanned documents, you’ll inevitably run into documents that are of poor quality. When that happens, your OCR accuracy will likely suffer.

Instead of throwing in the towel, consider how the techniques used in this chapter may help. Is it possible to manually pre-process a subset of these images and then use them as training data? From there, you can train a model that can accept a noisy pixel ROI and then produce a pristine, cleaned output.

Typically, we don’t use raw pixels as inputs to machine learning models (the exception being a convolutional neural network, of course). Usually, we’ll quantify an input image using some feature detector or descriptor extractor. From there, the resulting feature vector is handed off to a machine learning model.

Rarely does one see standard machine learning models operating on raw pixel intensities. It’s a neat trick that doesn’t feel like it should work in practice. *However, as you saw here, this method works!*

I hope you can use this chapter as a starting point when implementing your document denoising pipelines.

To go deeper, you could use denoising autoencoders to improve denoising quality. In this chapter, we used a random forest regressor, which essentially is an ensemble of different decision trees. Another ensemble you may want to explore is extreme gradient boosting or XGBoost, for short.



## Chapter 5

# Making OCR "Easy" with EasyOCR

So far, in this text, we have focused predominately on the Tesseract optical character recognition (OCR) engine. In a later chapter, we'll look at cloud-based OCR application programming interfaces (APIs), such as Amazon Rekognition API, Microsoft Cognitive Services, and Google Cloud Vision API.

All three of the cloud APIs mentioned above are **proprietary and require an internet connection**. The benefit of these APIs is that they can obtain higher OCR accuracy with less image pre-processing than what the Tesseract OCR engine typically does.

That said, is there a middle ground between the two? Is there another OCR engine that is:

- Locally installable (i.e., no internet connection required)
- Open-source (i.e., non-proprietary)
- Capable of obtaining higher OCR accuracy than Tesseract

There is such an OCR engine. **It's called EasyOCR.** And as the name suggests, EasyOCR is *by far* the most straightforward way to apply OCR:

- The EasyOCR package can be installed with a single `pip` command.
- The EasyOCR package's dependencies are minimal, making it easy to configure your OCR development environment.
- Once EasyOCR is installed, only one `import` statement is required to import the package into your project.
- From there, **all you need is two lines of code to perform OCR** — one to initialize the `Reader` class and then another to OCR the image via the `readtext` function.

Sound too good to be true?

Luckily, it's not — and in this chapter, I'll show you how to use EasyOCR to implement OCR in your projects.

## 5.1 Chapter Learning Objectives

In this chapter, you will:

- Learn about the EasyOCR package
- Configure your development environment to use EasyOCR
- Discover how easy it is to apply OCR using the EasyOCR package

## 5.2 Getting Started with EasyOCR

In the first part of this chapter, we'll briefly discuss the EasyOCR package. From there, we'll configure our OCR development environment and install EasyOCR on our machine.

Next, we'll implement a simple Python script that performs OCR via the EasyOCR package. You'll see firsthand how simple and straightforward it is to implement OCR (and even OCR text in *multiple languages*).

We'll wrap up this chapter with a discussion of the EasyOCR results.

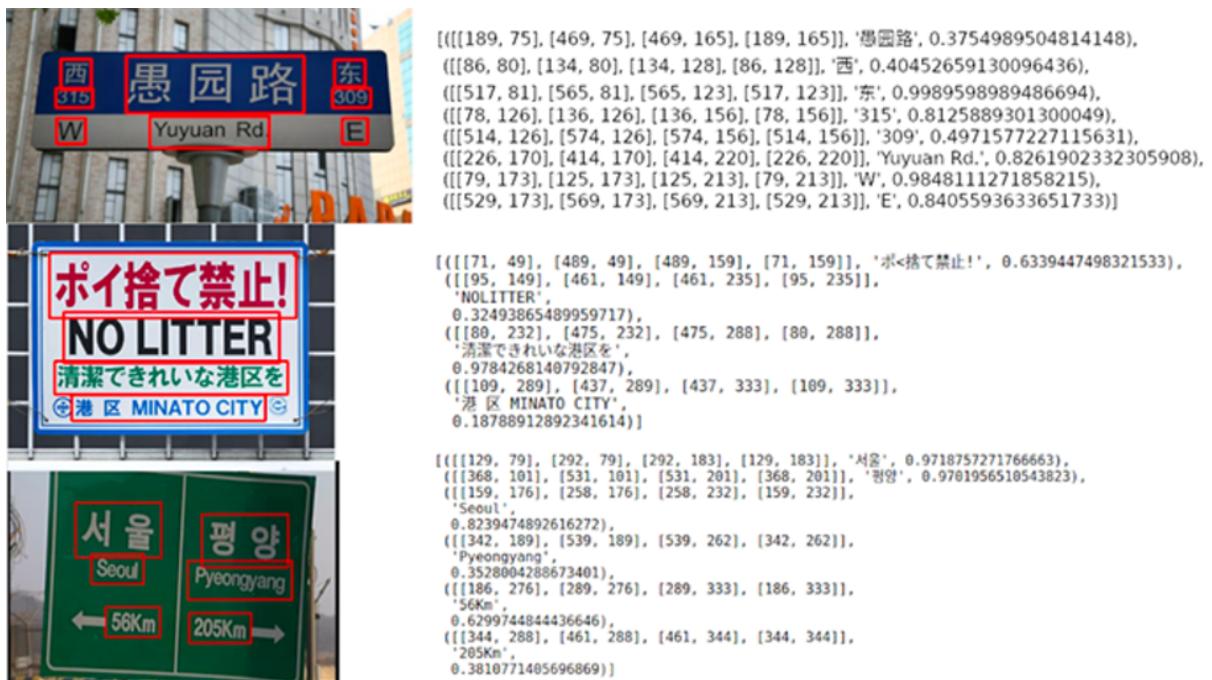
### 5.2.1 What Is EasyOCR?

The EasyOCR package (<http://pyimg.co/2w97h> [14]) is created and maintained by Jaided AI (<https://jaided.ai/> [15]), a company that specializes in OCR services.

EasyOCR is implemented using Python and the PyTorch library. If you have a CUDA-capable graphics processing unit (GPU), the underlying PyTorch deep learning library can *tremendously* speed up your text detection and OCR.

As of this writing, EasyOCR can OCR text in **58 languages**, including English, German, Hindi, Russian, *and more!* The EasyOCR maintainers plan to add additional languages in the future. You can find the full list of languages EasyOCR supports on the following page: <http://pyimg.co/90nj9> [16].

Currently, EasyOCR only supports OCR'ing typed text; however, according to their GitHub page, they plan to release handwriting recognition models as well! You can see EasyOCR in action in Figure 5.1.



**Figure 5.1.** Optical character recognition (OCR) is made easy with the EasyOCR Python package.

### 5.2.2 Installing EasyOCR

To install EasyOCR on your system, I recommend creating a **separate Python environment** from the ones you are currently using. I make this recommendation because EasyOCR requires several Python packages that may conflict with your other installs.

A primary example is if you have `opencv-contrib-python` installed on your machine. **Every PyImageSearch tutorial, book, or course recommends you use opencv-contrib-python, so if you followed one of my guides to install OpenCV on your system, it's more than likely that you have opencv-contrib-python installed.**

The problem here is that EasyOCR *instead* needs `opencv-python` — if you try to install both `opencv-python` and `opencv-contrib-python` into the *same* Python environment, your script will likely error out due to segmentation faults.

Instead, I recommend you create a brand new Python environment and then explicitly install `opencv-python`, `pyyaml`, and `easyocr` using the following commands:

---

```
$ pip install opencv-python # NOTE: *not* opencv-contrib-python
$ pip install pyyaml
$ pip install easyocr
```

---

From there, open a Python shell and validate that you can import both Python packages:

---

```
$ python
>>> import cv2
>>> import easyocr
>>>
```

---

Congrats on configuring your development environment for use with EasyOCR!

**Remark 5.1.** *If you are using the VM included with your purchase of this book, then you can use the pre-configured `easyocr` Python virtual environment instead of configuring your own. Just run the command `workon easyocr` to access it.*

## 5.3 Using EasyOCR for Optical Character Recognition

Now that we've discussed the EasyOCR package and installed it on our machine, we can move on to our implementation.

We'll start with a review of our project directory structure and then implement `easy_ocr.py`, a script that will allow us to easily OCR images using the EasyOCR package.

### 5.3.1 Project Structure

Let's inspect our project directory structure now:

---

```
|-- images
|   |-- arabic_sign.jpg
|   |-- swedish_sign.jpg
|   |-- turkish_sign.jpg
|-- pyimagesearch
|   |-- __init__.py
|   |-- helpers.py
|-- easy_ocr.py
```

---

This chapter's EasyOCR project is already appearing to live up to its name. As you can see, we have three testing `images/`, two Python files in `pyimagesearch/`, and a single Python driver script, `easy_ocr.py`. Our driver script accepts any input image and the desired OCR language to get the job done quite easily, as we'll see in the implementation section.

### 5.3.2 Implementing our EasyOCR Script

With our development environment configured and our project directory structure reviewed, we are now ready to use the EasyOCR package in our Python script!

Open the `easy_ocr.py` file in the project directory structure, and insert the following code:

---

```
1 # import the necessary packages
2 from pyimagesearch.helpers import cleanup_text
3 from easyocr import Reader
4 import argparse
5 import cv2
```

---

Our EasyOCR package should stand out here; notice how we're importing `Reader` from the `easyocr` package.

With our imports and convenience utility ready to go, let's now define our command line arguments:

---

```
7 # construct the argument parser and parse the arguments
8 ap = argparse.ArgumentParser()
9 ap.add_argument("-i", "--image", required=True,
10                 help="path to input image to be OCR'd")
11 ap.add_argument("-l", "--langs", type=str, default="en",
12                 help="comma separated list of languages to OCR")
13 ap.add_argument("-g", "--gpu", type=int, default=-1,
14                 help="whether or not GPU should be used")
15 args = vars(ap.parse_args())
```

---

Our script accepts three command line arguments:

- `--image`: The path to the input image containing text for OCR.
- `--langs`: A list of language codes separated by commas (no spaces). By `default`, our script assumes the English language (`en`). If you'd like to use the English and French models, you could pass `en, fr`. Or maybe you'd like to use Spanish, Portuguese, and Italian by using `es, pt, it`, respectively. Be sure to refer to EasyOCR's listing of supported languages: <http://pyimg.co/90nj9> [16].
- `--gpu`: Whether or not you'd like to use a GPU. Our `default` is `-1`, meaning that we'll use our central processing unit (CPU) rather than a GPU. If you have a CUDA-capable GPU, enabling this option will allow faster OCR results.

Given our command line arguments, let's now perform OCR:

---

```

17 # break the input languages into a comma separated list
18 langs = args["langs"].split(",")
19 print("[INFO] OCR'ing with the following languages: {}".format(langs))
20
21 # load the input image from disk
22 image = cv2.imread(args["image"])
23
24 # OCR the input image using EasyOCR
25 print("[INFO] OCR'ing input image...")
26 reader = Reader(langs, gpu=args["gpu"] > 0)
27 results = reader.readtext(image)

```

---

**Line 18** breaks our `--langs` string (comma-delimited) into a Python list of languages for our EasyOCR engine. We then load the input `--image` via **Line 22**.

**Note:** Unlike Tesseract, EasyOCR can work with OpenCV's default BGR (Blue Green Red) color channel ordering. Therefore, we do not need to swap color channels after loading the image.

To accomplish OCR with EasyOCR, we first instantiate a `Reader` object, passing the `langs` and `--gpu` Boolean to the constructor (**Line 26**). From there, we call the `readtext` method while passing our input image (**Line 27**).

Both the `Reader` class and `readtext` method are documented in the EasyOCR website (<http://pyimg.co/r24n1> [17]) if you'd like to customize your EasyOCR configuration.

Let's process our EasyOCR results now:

---

```

29 # loop over the results
30 for (bbox, text, prob) in results:
31     # display the OCR'd text and associated probability
32     print("[INFO] {:.4f}: {}".format(prob, text))
33
34     # unpack the bounding box
35     (tl, tr, br, bl) = bbox
36     tl = (int(tl[0]), int(tl[1]))
37     tr = (int(tr[0]), int(tr[1]))
38     br = (int(br[0]), int(br[1]))
39     bl = (int(bl[0]), int(bl[1]))
40
41     # cleanup the text and draw the box surrounding the text along
42     # with the OCR'd text itself
43     text = cleanup_text(text)
44     cv2.rectangle(image, tl, br, (0, 255, 0), 2)
45     cv2.putText(image, text, (tl[0], tl[1] - 10),
46                 cv2.FONT_HERSHEY_SIMPLEX, 0.8, (0, 255, 0), 2)
47
48 # show the output image

```

---

```
49 cv2.imshow("Image", image)
50 cv2.waitKey(0)
```

---

Our EasyOCR results consist of a 3-tuple:

- `bbox`: The bounding box coordinates of the localized text
- `text`: Our OCR'd string
- `prob`: The probability of the OCR results

Looping over each EasyOCR result (**Line 30**), we first unpack the bounding box coordinates (**Lines 35–39**). To prepare our `text` for annotation, we sanitize it via our `cleanup_text` utility (**Line 43**).

We then overlay our image with a bounding box surrounding the text and the `text` string itself (**Lines 44–46**).

After all the `results` are processed and annotated, **Lines 49 and 50** display the output image on our screen.

### 5.3.3 EasyOCR Results

We are now ready to see the results of applying OCR with the EasyOCR library.

Open a terminal and execute the following command:

```
$ python easy_ocr.py --image images/arabic_sign.jpg --langs en,ar
[INFO] OCR'ing with the following languages: ['en', 'ar']
[INFO] OCR'ing input image...
Using CPU. Note: This module is much faster with a GPU.
[INFO] 0.8129:
[INFO] 0.7237: EXIT
```

---

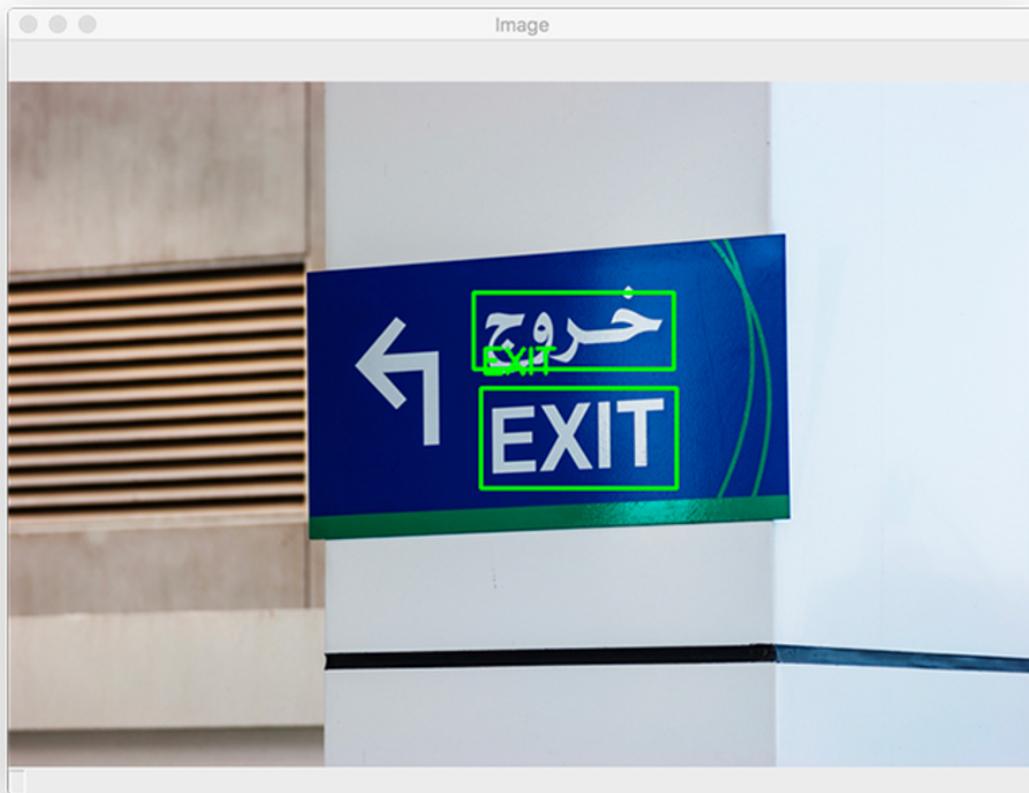
From the output of this command (Figure 5.2), you can see that I am OCR'ing an airport sign containing both English and Arabic text.

As the `--langs en,ar` arguments indicate, we're instructing our script (and ultimately EasyOCR) to OCR in Arabic and English. You may pass a comma-separated list of languages that EasyOCR supports: <http://pyimg.co/90nj9> [16].

EasyOCR can detect and correctly OCR the English and Arabic text in the input image.

**Note:** If you are using EasyOCR for the first time, you'll see an indication printed in your terminal that EasyOCR is "Downloading detection model[s]." Be patient while the files

download. Once these models are cached on your system, you can use them again and again seamlessly and quickly.



**Figure 5.2.** To get started with EasyOCR for OCR, let's try a picture of an “Exit” sign.

Let's try another image, this one containing a Swedish sign:

---

```
$ python easy_ocr.py --image images/swedish_sign.jpg --langs en,sv
[INFO] OCR'ing with the following languages: ['en', 'sv']
[INFO] OCR'ing input image...
Using CPU. Note: This module is much faster with a GPU.
[INFO] 0.7078: Fartkontrol
```

---

Figure 5.3 and the associated command above show that we ask EasyOCR to OCR both English (`en`) and Swedish (`sv`).

“*Fartkontrol*” is a bit of a joke amongst the Swedes and Danes. Translated, “*Fartkontrol*” in English means “*Speed Control*” (or simply speed monitoring).

However, when pronounced, “*Fartkontrol*” sounds like “*fart control*” — perhaps someone who

is having an issue controlling their flatulence. In college, I had a friend who hung a Swedish “*Fartkontrol*” sign on their bathroom door — maybe you don’t find the joke funny, but anytime I see that sign, I chuckle to myself (perhaps I’m just an immature 8-year-old).



**Figure 5.3.** Call me immature, but the Swedish translation of “*Speed Control*” looks an awful lot like “*Fart Control*.” If I get a speeding ticket in Sweden in my lifetime, I don’t think the traffic cop “*trafikpolis*” will find my jokes funny (Image Source: <http://pyimg.co/1lkum>).

For our final example, let’s look at a Turkish stop sign:

```
$ python easy_ocr.py --image images/turkish_sign.jpg --langs en,tr
[INFO] OCR'ing with the following languages: ['en', 'tr']
[INFO] OCR'ing input image...
Using CPU. Note: This module is much faster with a GPU.
[INFO] 0.9741: DUR
```

Here I ask EasyOCR to OCR both English (`en`) and Turkish (`tr`) texts by supplying those values as a comma-separated list via the `--langs` command line argument (Figure 5.4).

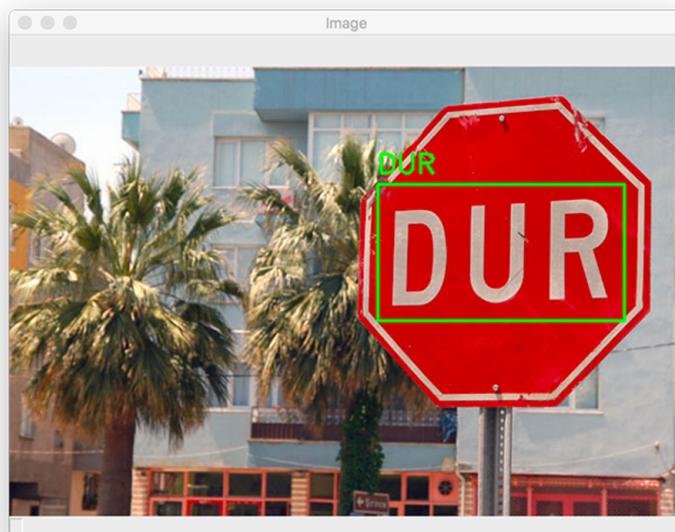
EasyOCR can detect the text, “*DUR*,” which, when translated from Turkish to English, is “*STOP*.”

As you can see, EasyOCR lives up to its name — finally, an easy-to-use OCR package!

Additionally, if you have a CUDA-capable GPU, you can obtain even faster OCR results by supplying the `--gpu` command line argument, as in the following:

```
$ python easy_ocr.py --image images/turkish_sign.jpg --langs en,tr --gpu 1
```

But again, you will need to have a CUDA GPU configured for the PyTorch library (EasyOCR uses the PyTorch deep learning library under the hood).



**Figure 5.4.** The Turkish translation for “STOP” is properly OCR’d as “DUR” (image source: <http://pyimg.co/2ta42>).

## 5.4 Summary

In this chapter, you learned how to perform OCR using the EasyOCR Python package.

Unlike the Tesseract OCR engine and the `pytesseract` package, which can be a bit tedious to work with if you are new to the world of OCR, the EasyOCR package lives up to its name — **EasyOCR makes OCR with Python “easy.”**

Furthermore, EasyOCR has many benefits going for it:

- i. You can use your GPU to increase the speed of your OCR pipeline.
- ii. You can use EasyOCR to OCR text in *multiple languages at the same time*.
- iii. The EasyOCR API is Pythonic, making it simple and intuitive to use.

When developing your OCR applications, definitely give EasyOCR a try. It could save you much time and headache, *especially* if you look for an OCR engine that is entirely `pip`-installable with no other dependencies or if you wish to apply OCR to multiple languages.

## Chapter 6

# Image/Document Alignment and Registration

In this chapter, you will learn how to perform image alignment and image registration using OpenCV. Image alignment and registration have several practical, real-world use cases, including:

- **Medical:** Magnetic resonance imaging (MRI) scans, single-photon emission computerized tomography (SPECT) scans, and other medical scans produce multiple images. Image registration can *align* multiple images together and overlay them on top of each other to help doctors and physicians better interpret these scans. From there, the doctor can read the results and provide a more accurate diagnosis.
- **Military:** Automatic target recognition (ATR) algorithms accept multiple input images of the target, align them, and refine their internal parameters to improve target recognition.
- **Optical character recognition (OCR):** Image alignment (often called document alignment in the context of OCR) can be used to build automatic form, invoice, or receipt scanners. We first align the input image to a template of the document we want to scan. From there, OCR algorithms can read the text from each field.

In this chapter's context, we'll be looking at image alignment through the perspective of **document alignment/registration**, which is often used in OCR applications.

This chapter will cover the fundamentals of image registration and alignment. Then, in the next chapter, we'll incorporate image alignment with OCR, allowing us to create a document, form, and invoice scanner that aligns an input image with a template document and then extracts the text from each field in the document.

## 6.1 Chapter Learning Objectives

In this chapter, you will:

- Learn how to detect keypoints and extract local invariant descriptors using oriented FAST and rotated BRIEF (ORB)
- Discover how to match keypoints together and draw correspondences
- Compute a homography matrix using the correspondences
- Take the homography matrix and use it to apply a perspective warp, thereby aligning the two images/documents together

## 6.2 Document Alignment and Registration with OpenCV

In the first part of this chapter, we'll briefly discuss what image alignment and registration is. We'll learn how OpenCV can help us align and register our images using keypoint detectors, local invariant descriptors, and keypoint matching.

Next, we'll implement a helper function, `align_images`, which, as the name suggests, will allow us to align two images based on keypoint correspondences.

I'll then show you how to use the `align_document` function to align an input image with a template.

### 6.2.1 What Is Document Alignment and Registration?

Image alignment and registration is the process of:

- i. Accepting two input images that contain the *same* object but at slightly different *viewing angles*
- ii. Automatically computing the homography matrix used to align the images (whether that be feature-based keypoint correspondences, similarity measures, or even deep neural networks that *automatically* learn the transformation)
- iii. Taking that homography matrix and applying a perspective warp to align the images together

For example, consider Figure 6.1. On the *top-left*, we have a template of a W-4 form, which is a U.S. Internal Revenue Service (IRS) tax form that employees fill out so that employers know how much tax to withhold from their paycheck (depending on deductions, filing status, etc.).

## *6.2. Document Alignment and Registration with OpenCV*

73

<b>Form W-4</b> Department of the Treasury Internal Revenue Service	<b>Employee's Withholding Certificate</b> OMB No. 1545-0074 ► Complete Form W-4 so that your employer can withhold the correct federal income tax from your pay. ► Give Form W-4 to your employer. ► Your withholding is subject to review by the IRS.	<b>2020</b>
<p><b>Step 1:</b> (a) First name and middle initial      Last name      (b) Social security number</p> <p>Address</p> <p>City or town, state, and ZIP code</p> <p>(c) <input type="checkbox"/> Single or Married filing separately  <input type="checkbox"/> Married filing jointly (or Qualifying widow(er))  <input type="checkbox"/> Head of household (Check only if you're unmarried and pay more than half the costs of keeping up a home for yourself and a qualifying individual.)</p> <p><b>Complete Steps 2-4 ONLY if they apply to you; otherwise, skip to Step 5.</b> See page 2 for more information on each step, who can claim exemption from withholding, when to use the online estimator, and privacy.</p> <p><b>Step 2:</b> <b>Multiple Jobs or Spouse Works</b>          Complete this step if you (1) hold more than one job at a time, or (2) are married filing jointly and your spouse does only one of the following:          (a) Use the estimator at <a href="http://www.irs.gov/W4App">www.irs.gov/W4App</a> for most accurate withholding for this step (and Steps 3-4); or          (b) Use the Multiple Jobs Worksheet on page 3 and enter the result in Step 4(c) below for roughly accurate withholding; or          (c) If there are only two jobs total, you may check this box. Do the same on Form W-4 for the other job. This option is accurate for jobs with similar pay; otherwise, more tax than necessary may be withheld.      ▶ <b>TIP:</b> To be accurate, submit a 2020 Form W-4 for all other jobs. If you (or your spouse) have self-employment income, including as an independent contractor, use the estimator.</p> <p><b>Complete Steps 3-4(b) on Form W-4 for only ONE of these jobs.</b> Leave those steps blank for the other jobs. (Your withholding will be most accurate if you complete Steps 3-4(b) on the Form W-4 for the highest paying job.)</p> <p><b>Step 3:</b> <b>Claim Dependents</b>          If your income will be \$200,000 or less (\$400,000 or less if married filing jointly):          Multiply the number of qualifying children under age 17 by \$2,000 ► <b>\$</b>          Multiply the number of other dependents by \$500      ▶ <b>\$</b>          Add the amounts above and enter the total here      <b>3</b> <b>\$</b></p> <p><b>Step 4 (optional): Other Adjustments</b>          (a) <b>Other income (not from jobs).</b> If you want tax withheld for other income you expect this year that won't have withholding, enter the amount of other income here. This may include interest, dividends, and retirement income      <b>4(a)</b> <b>\$</b>          (b) <b>Deductions.</b> If you expect to claim deductions other than the standard deduction and want to reduce your withholding, use the Deductions Worksheet on page 3 and enter the result here      <b>4(b)</b> <b>\$</b>          (c) <b>Extra withholding.</b> Enter any additional tax you want withheld each pay period      <b>4(c)</b> <b>\$</b></p> <p><b>Step 5:</b> <b>Sign Here</b>          Under penalties of perjury, I declare that this certificate, to the best of my knowledge and belief, is true, correct, and complete.          Employee's signature (This form is not valid unless you sign it.)      Date</p> <p><b>Employers Only</b> Employer's name and address      First date of employment      Employer identification number (EIN)</p> <p><b>For Privacy Act and Paperwork Reduction Act Notice, see page 3.</b> Cat. No. 10220Q Form W-4 (2020)</p>		
<p><b>Form W-4</b>          Department of the Treasury          Internal Revenue Service</p> <p>► Complete Form W-4 so that your employer can withhold the correct federal income tax from your pay.          ► Give Form W-4 to your employer.          ► Your withholding is subject to review by the IRS.</p> <p><b>2020</b></p> <p><b>Employee's Withholding Certificate</b>          OMB No. 1545-0074          ► Complete Form W-4 so that your employer can withhold the correct federal income tax from your pay.          ► Give Form W-4 to your employer.          ► Your withholding is subject to review by the IRS.</p> <p><b>Step 1:</b> <b>Enter Personal Information</b>          PO Box 17598 #17600          City or town, state, and ZIP code          MI, Adrian, MI 21927-1598</p> <p><b>Step 2:</b> <b>Multiple Jobs or Spouse Works</b>          Complete this step if you (1) hold more than one job at a time, or (2) are married filing jointly and your spouse does only one of the following:          (a) Use the estimator at <a href="http://www.irs.gov/W4App">www.irs.gov/W4App</a> for most accurate withholding for this step (and Steps 3-4); or          (b) Use the Multiple Jobs Worksheet on page 3 and enter the result in Step 4(c) below for roughly accurate withholding; or          (c) If there are only two jobs total, you may check this box. Do the same on Form W-4 for the other job. This option is accurate for jobs with similar pay; otherwise, more tax than necessary may be withheld.      ▶ <b>TIP:</b> To be accurate, submit a 2020 Form W-4 for all other jobs. If you (or your spouse) have self-employment income, including as an independent contractor, use the estimator.</p> <p><b>Complete Steps 3-4(b) on Form W-4 for only ONE of these jobs.</b> Leave those steps blank for the other jobs. (Your withholding will be most accurate if you complete Steps 3-4(b) on the Form W-4 for the highest paying job.)</p> <p><b>Step 3:</b> <b>Claim Dependents</b>          Multiply the number of qualifying children under age 17 by \$2,000 ► <b>\$</b>          Multiply the number of other dependents by \$500      ▶ <b>\$</b>          Add the amounts above and enter the total here      <b>3</b> <b>\$</b></p> <p><b>Step 4 (optional): Other Adjustments</b>          (a) <b>Other income (not from jobs).</b> If you want tax withheld for other income you expect this year that won't have withholding, enter the amount of other income here. This may include interest, dividends, and retirement income      <b>4(a)</b> <b>\$</b>          (b) <b>Deductions.</b> If you expect to claim deductions other than the standard deduction and want to reduce your withholding, use the Deductions Worksheet on page 3 and enter the result here      <b>4(b)</b> <b>\$</b>          (c) <b>Extra withholding.</b> Enter any additional tax you want withheld each pay period      <b>4(c)</b> <b>\$</b></p> <p><b>Step 5:</b> <b>Sign Here</b>          Under penalties of perjury, I declare that this certificate, to the best of my knowledge and belief, is true, correct, and complete.          Employee's signature (This form is not valid unless you sign it.)      Date</p> <p><b>Employers Only</b> Employer's name and address          PyImageSearch          PO BOX 1234          Philadelphia, PA 19019          First date of employment          Employer identification number (EIN)          12-3456789</p> <p><b>For Privacy Act and Paperwork Reduction Act Notice, see page 3.</b> Cat. No. 10220Q Form W-4 (2020)</p>		
<p><b>Form W-4</b>          Department of the Treasury          Internal Revenue Service</p> <p>► Complete Form W-4 so that your employer can withhold the correct federal income tax from your pay.          ► Give Form W-4 to your employer.          ► Your withholding is subject to review by the IRS.</p> <p><b>2020</b></p> <p><b>Employee's Withholding Certificate</b>          OMB No. 1545-0074          ► Complete Form W-4 so that your employer can withhold the correct federal income tax from your pay.          ► Give Form W-4 to your employer.          ► Your withholding is subject to review by the IRS.</p> <p><b>Step 1:</b> <b>Enter Personal Information</b>          PO Box 17598 #17600          City or town, state, and ZIP code          MI, Adrian, MI 21927-1598</p> <p><b>Step 2:</b> <b>Multiple Jobs or Spouse Works</b>          Complete this step if you (1) hold more than one job at a time, or (2) are married filing jointly and your spouse does only one of the following:          (a) Use the estimator at <a href="http://www.irs.gov/W4App">www.irs.gov/W4App</a> for most accurate withholding for this step (and Steps 3-4); or          (b) Use the Multiple Jobs Worksheet on page 3 and enter the result in Step 4(c) below for roughly accurate withholding; or          (c) If there are only two jobs total, you may check this box. Do the same on Form W-4 for the other job. This option is accurate for jobs with similar pay; otherwise, more tax than necessary may be withheld.      ▶ <b>TIP:</b> To be accurate, submit a 2020 Form W-4 for all other jobs. If you (or your spouse) have self-employment income, including as an independent contractor, use the estimator.</p> <p><b>Complete Steps 3-4(b) on Form W-4 for only ONE of these jobs.</b> Leave those steps blank for the other jobs. (Your withholding will be most accurate if you complete Steps 3-4(b) on the Form W-4 for the highest paying job.)</p> <p><b>Step 3:</b> <b>Claim Dependents</b>          If your income will be \$200,000 or less (\$400,000 or less if married filing jointly):          Multiply the number of qualifying children under age 17 by \$2,000 ► <b>\$</b>          Multiply the number of other dependents by \$500      ▶ <b>\$</b>          Add the amounts above and enter the total here      <b>3</b> <b>\$</b></p> <p><b>Step 4 (optional): Other Adjustments</b>          (a) <b>Other income (not from jobs).</b> If you want tax withheld for other income you expect this year that won't have withholding, enter the amount of other income here. This may include interest, dividends, and retirement income      <b>4(a)</b> <b>\$</b>          (b) <b>Deductions.</b> If you expect to claim deductions other than the standard deduction and want to reduce your withholding, use the Deductions Worksheet on page 3 and enter the result here      <b>4(b)</b> <b>\$</b>          (c) <b>Extra withholding.</b> Enter any additional tax you want withheld each pay period      <b>4(c)</b> <b>\$</b></p> <p><b>Step 5:</b> <b>Sign Here</b>          Under penalties of perjury, I declare that this certificate, to the best of my knowledge and belief, is true, correct, and complete.          Employee's signature (This form is not valid unless you sign it.)      Date</p> <p><b>Employers Only</b> Employer's name and address          PyImageSearch          PO BOX 1234          Philadelphia, PA 19019          First date of employment          Employer identification number (EIN)          12-3456789</p> <p><b>For Privacy Act and Paperwork Reduction Act Notice, see page 3.</b> Cat. No. 10220Q Form W-4 (2020)</p>		

**Figure 6.1.** We have three scans of the well-known IRS W-4 form. We have an empty form (*top-left*), a partially completed form that is out of alignment (*top-right*), and our processed form that has been aligned by our algorithm (*bottom*).

I have partially filled out a W-4 form with faux data and then captured a photo of it with my phone (*top-right*).

Finally, you can see the output image alignment and registration on the bottom — **notice how the input image is now aligned with the template.**

In the next chapter, you'll learn how to OCR each of the individual fields from the input document and associate them with the template fields. For now, though, we'll only be learning how to align a form with its template as an important pre-processing step before applying OCR.

### 6.2.2 How Can OpenCV Be Used for Document Alignment?

There are several image alignment and registration algorithms:

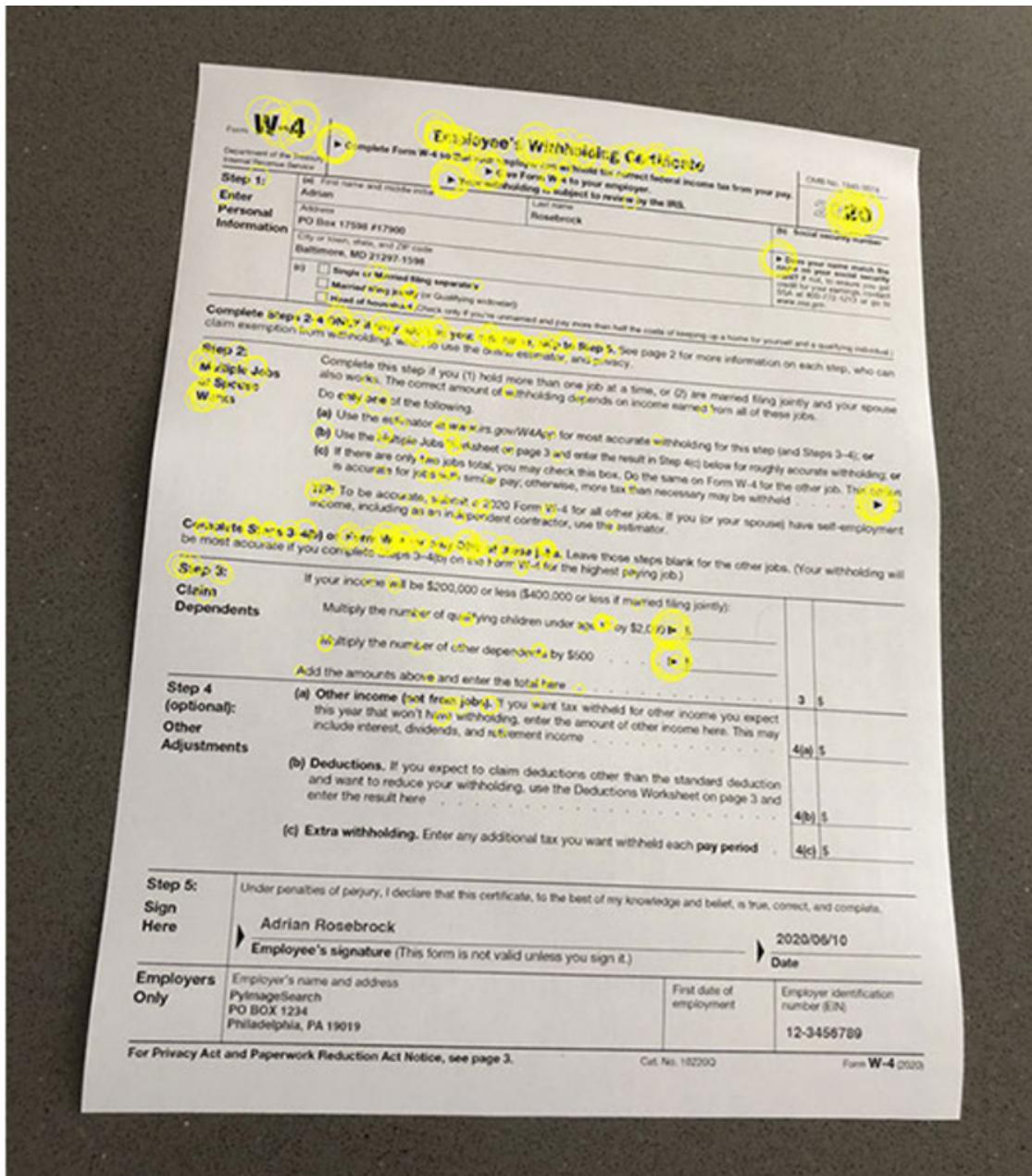
- The most popular image alignment algorithms are **feature-based**. They include keypoint detectors (DoG, Harris, good features to track, etc.), local invariant descriptors (SIFT, SURF, ORB, etc.), and keypoint matching such as random sample consensus (RANSAC) and its variants.
- Medical applications often use **similarity measures** for image registration, typically cross-correlation, the sum of squared intensity differences, and mutual information.
- With the resurgence of neural networks, **deep learning** can even be used for image alignment by automatically learning the homography transform.

We'll be implementing image alignment and registration using feature-based methods.

Feature-based methods start with detecting keypoints in our two input images (Figure 6.2).

Keypoints are meant to identify salient regions of an input image. We extract local invariant descriptors for each keypoint, quantifying the region surrounding each keypoint in the input image.

Scale-invariant feature transform (SIFT) features, for example, are 128-d, so if we detect 528 keypoints in a given input image, then we'll have a total of 528 vectors, each of which is 128-d.

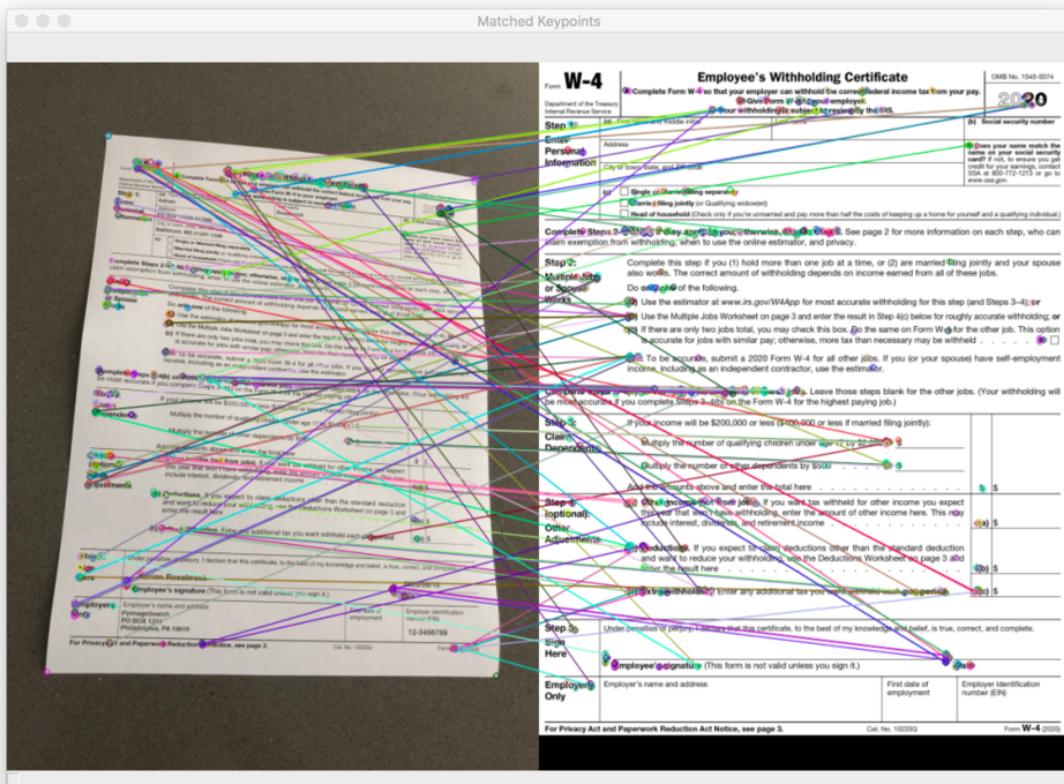


**Figure 6.2.** We use a feature-based image alignment technique that detects keypoints that are the location of regions of interest. The keypoints found are highlighted in yellow. They will be used to align and register the two images (documents/forms).

Given our features, we apply algorithms such as RANSAC [18] to match our keypoints and determine their correspondences (Figure 6.3).

Provided we have enough keypoint matches and correspondences. We can then compute a homography matrix, which allows us to apply a perspective warp to align the images. You'll be learning how to build an OpenCV project that accomplishes image alignment and registration via a homography matrix in the remainder of this chapter.

For more details on homography matrix construction and the role it plays in computer vision, be sure to refer to this OpenCV reference: <http://pyimg.co/ogkky> [19].



**Figure 6.3.** Here we see the keypoints matched between our partially completed form that is out of alignment with our original form that is properly aligned. You can see there are colored lines drawn using OpenCV between each corresponding keypoint pair.

### 6.2.3 Project Structure

Let's start by inspecting our project directory structure:

```

|-- pyimagesearch
|   |-- __init__.py
|   |-- alignment
|       |-- __init__.py
|       |-- align_images.py
|-- forms
|   |-- form_w4.pdf
|   |-- form_w4_orig.pdf
|-- scans
|   |-- scan_01.jpg
|   |-- scan_02.jpg

```

```
|-- align_document.py  
|-- form_w4.png
```

---

We have a simple project structure consisting of the following images:

- `scans/`: Contains two JPG testing photos of a tax form
- `form_w4.png`: Our template image of the official 2020 IRS W-4 form

Additionally, we'll be reviewing two Python files:

- `align_images.py`: Holds our helper function, which aligns a scan to a template utilizing an OpenCV pipeline
- `align_document.py`: Our driver file in the main directory brings all the pieces together to perform image alignment and registration with OpenCV

In the next section, we'll work on implementing our helper utility for aligning images.

#### 6.2.4 Aligning Documents with Keypoint Matching

We are now ready to implement image alignment and registration using OpenCV. For this section, we'll attempt to align the images in Figure 6.4.

On the *left*, we have our template W-4 form, while on the *right*, we have a sample W-4 form I have filled out and captured with my phone. The end goal is to align these images such that their fields match up.

Let's get started! Open `align_images.py` inside the `alignment` submodule of `pyimagesearch` and let's get to work:

---

```
1 # import the necessary packages  
2 import numpy as np  
3 import imutils  
4 import cv2  
5  
6 def align_images(image, template, maxFeatures=500, keepPercent=0.2,  
7     debug=False):  
8     # convert both the input image and template to grayscale  
9     imageGray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)  
10    templateGray = cv2.cvtColor(template, cv2.COLOR_BGR2GRAY)
```

---

**W-4**

**Employee's Withholding Certificate**

Form W-4  
Department of the Treasury  
Internal Revenue Service

CHUB No. 1545-0074  
2020

**Step 1: Enter Personal Information**

(a) First name and middle initial      Last name      (b) Social security number

Address  
City or town, state, and ZIP code

(c) Single or Married filing jointly  
Married filing jointly (or Qualifying widow)  
Head of household (Check only if you're unmarried and pay more than half the costs of keeping up a home for yourself and a qualifying individual.)

**Step 2:** Complete this step if you (1) hold more than one job at a time, or (2) are married filing jointly and your spouse also works. The correct amount of withholding depends on income earned from all of these jobs.  
Do only one of the following:  
(a) Use the estimator at [www.irs.gov/W4Agp](http://www.irs.gov/W4Agp) for most accurate withholding for this step (and Steps 3-4); or  
(b) Use the Multiple Jobs Worksheet on page 3 and enter the result in Step 4(d) below for roughly accurate withholding; or  
(c) If there are only two jobs total, you may check this box. Do the same on Form W-4 for the other job. This option is accurate for jobs with similar pay; otherwise, more tax than necessary may be withheld.

**Step 3:** Complete Steps 2-4 ONLY if they apply to you; otherwise, skip to Step 5. See page 2 for more information on each step, who can claim exemption from withholding, when to use the online estimator, and privacy.

**Step 4:** (optional) Other Adjustments

(a) Other income (not from jobs). If you want tax withheld for other income you expect this year that won't have withholding, enter the amount of other income here. This may include interest, dividends, and retirement income.

(b) Deductions. If you expect to claim deductions other than the standard deduction and want to reduce your withholding, use the Deductions Worksheet on page 3 and enter the result here.

(c) Extra withholding. Enter any additional tax you want withheld each pay period.

**Step 5: Sign Here**

Under penalties of perjury, I declare that this certificate, to the best of my knowledge and belief, is true, correct, and complete.

Employee's signature (This form is not valid unless you sign it.)      Date

Employer's Name and address  
First date of employment  
Employee identification number (SIN)

For Privacy Act and Paperwork Reduction Act Notice, see page 3.

Form W-4 (2020)      Cat. No. 15220D      Form W-4 (2020)

**W-4**

**Employee's Withholding Certificate**

Form W-4  
Department of the Treasury  
Internal Revenue Service

CHUB No. 1545-0074  
2020

**Step 1: Enter Personal Information**

(a) First name and middle initial      Last name      (b) Social security number

Address  
City or town, state, and ZIP code

(c) Single or Married filing jointly  
Married filing jointly (or Qualifying widow)  
Head of household (Check only if you're unmarried and pay more than half the costs of keeping up a home for yourself and a qualifying individual.)

**Step 2:** Complete this step if you (1) hold more than one job at a time, or (2) are married filing jointly and your spouse also works. The correct amount of withholding depends on income earned from all of these jobs.  
Do only one of the following:  
(a) Use the estimator at [www.irs.gov/W4Agp](http://www.irs.gov/W4Agp) for most accurate withholding for this step (and Steps 3-4); or  
(b) Use the Multiple Jobs Worksheet on page 3 and enter the result in Step 4(d) below for roughly accurate withholding; or  
(c) If there are only two jobs total, you may check this box. Do the same on Form W-4 for the other job. This option is accurate for jobs with similar pay; otherwise, more tax than necessary may be withheld.

**Step 3:** Complete Steps 2-4 ONLY if they apply to you; otherwise, skip to Step 5. See page 2 for more information on each step, who can claim exemption from withholding, when to use the online estimator, and privacy.

**Step 4:** (optional) Other Adjustments

(a) Other income (not from jobs). If you want tax withheld for other income you expect this year that won't have withholding, enter the amount of other income here. This may include interest, dividends, and retirement income.

(b) Deductions. If you expect to claim deductions other than the standard deduction and want to reduce your withholding, use the Deductions Worksheet on page 3 and enter the result here.

(c) Extra withholding. Enter any additional tax you want withheld each pay period.

**Step 5: Sign Here**

Under penalties of perjury, I declare that this certificate, to the best of my knowledge and belief, is true, correct, and complete.

Employee's signature (This form is not valid unless you sign it.)      Date

Employer's Name and address  
First date of employment  
Employee identification number (SIN)

For Privacy Act and Paperwork Reduction Act Notice, see page 3.

Form W-4 (2020)      Cat. No. 15220D      Form W-4 (2020)

**Figure 6.4.** We have two similar W-4 forms in different orientations. We have our original template form in the desired orientation (*left*) and the partially completed form that needs to be aligned (*right*). Our goal is to use OpenCV to align the *right* image to the *left* template image using keypoint matching and a homography matrix.

Our `align_images` function begins on **Line 6** and accepts five parameters:

- `image`: Our input photo/scan of a form (e.g., IRS W-4). From an arbitrary viewpoint, the form itself should be identical to the `template` image but with form data present.
- `template`: The template form image.
- `maxFeatures`: Places an upper bound on the number of candidate keypoint regions to consider.
- `keepPercent`: Designates the percentage of keypoint matches to keep, effectively allowing us to eliminate noisy keypoint matching results
- `debug`: A flag indicating whether to display the matched keypoints. By default, keypoints are not displayed. However, I recommend setting this value to `True` for debugging purposes.

Given that we have defined our function let's implement our image processing pipeline. Diving in, the first action we take is converting both our `image` and `template` to grayscale (**Lines 9 and 10**).

Next, we will detect keypoints, extract local binary features, and correlate these features between our input image and the template:

---

```

12     # use ORB to detect keypoints and extract (binary) local
13     # invariant features
14     orb = cv2.ORB_create(maxFeatures)
15     (kpsA, descsA) = orb.detectAndCompute(imageGray, None)
16     (kpsB, descsB) = orb.detectAndCompute(templateGray, None)
17
18     # match the features
19     method = cv2.DESCRIPTOR_MATCHER_BRUTEFORCE_HAMMING
20     matcher = cv2.DescriptorMatcher_create(method)
21     matches = matcher.match(descsA, descsB, None)

```

---

We use the ORB algorithm (<http://pyimg.co/46f2s> [20]) to detect keypoints and extract binary local invariant features (**Lines 14–16**). The Hamming method computes the distance between these binary features to find the best matches (**Lines 19–21**). You can learn more about the keypoint detection and local binary patterns in my *Local Binary Patterns with Python & OpenCV* tutorial (<http://pyimg.co/94p11> [21]) or the PyImageSearch Gurus course (<http://pyimg.co/gurus> [22]).

As we now have our keypoint `matches`, our next steps include sorting, filtering, and displaying:

---

```

23     # sort the matches by their distance (the smaller the distance,
24     # the "more similar" the features are)
25     matches = sorted(matches, key=lambda x:x.distance)
26
27     # keep only the top matches
28     keep = int(len(matches) * keepPercent)
29     matches = matches[:keep]
30
31     # check to see if we should visualize the matched keypoints
32     if debug:
33         matchedVis = cv2.drawMatches(image, kpsA, template, kpsB,
34             matches, None)
35         matchedVis = imutils.resize(matchedVis, width=1000)
36         cv2.imshow("Matched Keypoints", matchedVis)
37         cv2.waitKey(0)

```

---

Here, we sort the `matches` (**Line 25**) by their distance. The *smaller* the distance, the *more similar* the two keypoint regions are. **Lines 28 and 29** keep only the top matches — otherwise we risk introducing noise.

If we are in `debug` mode, we will use `cv2.drawMatches` to visualize the matches using OpenCV drawing methods (**Lines 32–37**).

Next, we will conduct a couple of steps before computing our homography matrix:

---

```

39      # allocate memory for the keypoints (x,y-coordinates) from the
40      # top matches -- we'll use these coordinates to compute our
41      # homography matrix
42      ptsA = np.zeros((len(matches), 2), dtype="float")
43      ptsB = np.zeros((len(matches), 2), dtype="float")
44
45      # loop over the top matches
46      for (i, m) in enumerate(matches):
47          # indicate that the two keypoints in the respective images
48          # map to each other
49          ptsA[i] = kpsA[m.queryIdx].pt
50          ptsB[i] = kpsB[m.trainIdx].pt

```

---

Here we are:

- Allocating memory to store the keypoints (`ptsA` and `ptsB`) for our top matches
- Initiating a loop over our `top_matches` and inside indicating that keypoints A and B map to one another

Given our organized pairs of keypoint matches, now we're ready to align our image:

---

```

52      # compute the homography matrix between the two sets of matched
53      # points
54      (H, mask) = cv2.findHomography(ptsA, ptsB, method=cv2.RANSAC)
55
56      # use the homography matrix to align the images
57      (h, w) = template.shape[:2]
58      aligned = cv2.warpPerspective(image, H, (w, h))
59
60      # return the aligned image
61      return aligned

```

---

Aligning our image can be boiled down to our final two steps:

- Find our homography matrix using the keypoints and RANSAC algorithm (**Line 54**).
- Align our image by applying a warped perspective (`cv2.warpPerspective`) to our image and matrix, `H` (**Lines 57 and 58**). This aligned image result is returned to the caller via **Line 61**.

Congratulations! You have completed the most technical part of the chapter.

**Note:** A big thanks to Satya Mallick at LearnOpenCV for his concise implementation of keypoint matching (<http://pyimg.co/16qi2> [23]), upon which ours is based.

### 6.2.5 Implementing Our Document Alignment Script

Now that we have the `align_images` function at our disposal, we need to develop a driver script that:

- i. Loads an image and template from disk
- ii. Performs image alignment and registration
- iii. Displays the aligned images to our screen to verify that our image registration process is working properly

Open `align_document.py`, and let's review it to see how we can accomplish exactly that:

---

```

1 # import the necessary packages
2 from pyimagesearch.alignment import align_images
3 import numpy as np
4 import argparse
5 import imutils
6 import cv2
7
8 # construct the argument parser and parse the arguments
9 ap = argparse.ArgumentParser()
10 ap.add_argument("-i", "--image", required=True,
11     help="path to input image that we'll align to template")
12 ap.add_argument("-t", "--template", required=True,
13     help="path to input template image")
14 args = vars(ap.parse_args())

```

---

The key import on **Lines 2–6** that should stand out is `align_images`, a function which we implemented in the previous section.

Our script requires two command line arguments:

- `--image`: The path to the input image scan or photo
- `--template`: Our template image path; this could be an official company form or in our case the 2020 IRS W-4 template image

Our next step is to align our two input images:

---

```

16 # load the input image and template from disk
17 print("[INFO] loading images...")
18 image = cv2.imread(args["image"])
19 template = cv2.imread(args["template"])
20

```

---

---

```

21 # align the images
22 print("[INFO] aligning images...")
23 aligned = align_images(image, template, debug=True)

```

---

After loading both our input `--image` and input `--template` (**Lines 18 and 19**), we take advantage of our helper routine, `align_images`, passing each as a parameter (**Line 23**).

Notice how I've set the `debug` flag to `True`, indicating that I'd like the matches to be annotated. When you make the `align_images` function part of a real OCR pipeline you will turn the debugging option off.

We're going to visualize our results in two ways:

- i. **Stacked** side-by-side
- ii. **Overlaid** on top of one another

These visual representations of our results will allow us to determine if the alignment is/was successful.

Let's prepare our `aligned` image for a *stacked* comparison with its template:

---

```

25 # resize both the aligned and template images so we can easily
26 # visualize them on our screen
27 aligned = imutils.resize(aligned, width=700)
28 template = imutils.resize(template, width=700)
29
30 # our first output visualization of the image alignment will be a
31 # side-by-side comparison of the output aligned image and the
32 # template
33 stacked = np.hstack([aligned, template])

```

---

**Lines 27 and 28** resize the two images such that they will fit on our screen. We then use `np.hstack` to *stack our images next to each other* to easily inspect the results (**Line 33**).

And now let's *overlay* the `template` form on top of the `aligned` image:

---

```

35 # our second image alignment visualization will be *overlays* the
36 # aligned image on the template, that way we can obtain an idea of
37 # how good our image alignment is
38 overlay = template.copy()
39 output = aligned.copy()
40 cv2.addWeighted(overlay, 0.5, output, 0.5, 0, output)
41
42 # show the two output image alignment visualizations
43 cv2.imshow("Image Alignment Stacked", stacked)

```

---

```
44 cv2.imshow("Image Alignment Overlay", output)
45 cv2.waitKey(0)
```

---

In addition to the side-by-side stacked visualization from above, an alternative visualization is to *overlay* the input image on the template, so we can readily see the amount of misalignment. **Lines 38–40** use OpenCV’s `cv2.addWeighted` to transparently blend the two images into a single `output` image with the pixels from each image having equal weight.

Finally, we display our two visualizations on-screen (**Lines 43–45**).

Well done! It is now time to inspect our results.

### 6.2.6 Image and Document Alignment Results

We are now ready to apply image alignment and registration using OpenCV! Open a terminal and execute the following command:

```
$ python align_document.py --template form_w4.png --image scans/scan_01.jpg
[INFO] loading images...
[INFO] aligning images...
```

---

Figure 6.5 (*top*) shows our input image, `scan_01.jpg` — notice how this image has been captured with my smartphone at a non-90° viewing angle (i.e., not a *top-down*, bird’s-eye view of the input image).

It’s very common for images to be captured from unpredictable angles when capturing documents with phone cameras.

To get better light, people will often move to the side of an image creating an angle to their photo.

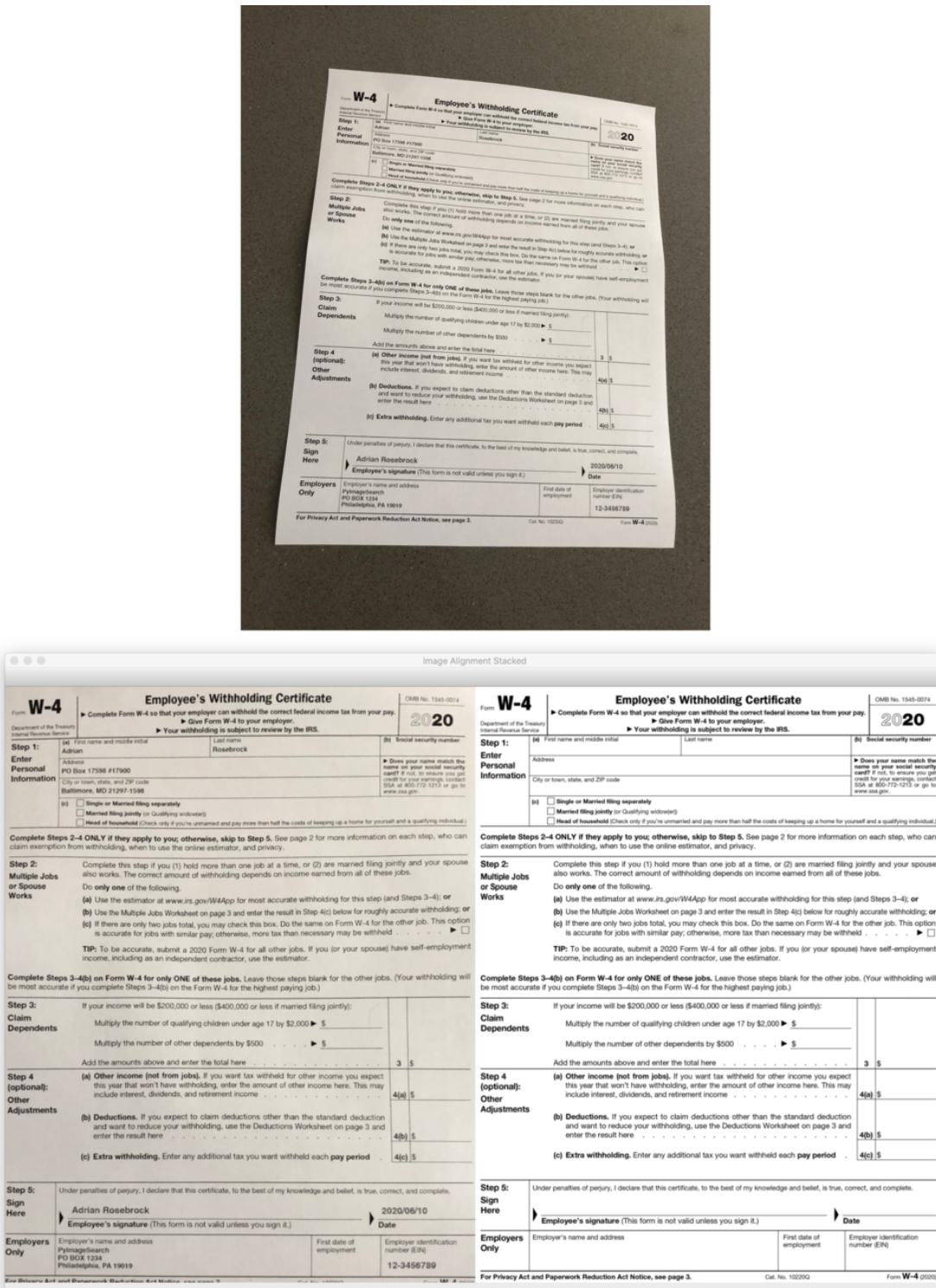
Many times people won’t even be aware they are taking a picture at an angle. This means many images are captured from an angle. For OCR to work, we need a way to allow people to take photos from multiple angles. We are the ones who need to align the images after they are taken.

To create an OCR pipeline that doesn’t do this alignment would result in a brittle and inaccurate system.

We then apply image alignment and registration, resulting in Figure 6.5 (*bottom*).

On the *bottom-left*, you can see the input image (after alignment), while the *bottom-right* shows the original W-4 template image. Notice how the two images have been automatically aligned using keypoint matching!

## Chapter 6. Image/Document Alignment and Registration



**Figure 6.5. Top:** We have our partially completed form, which has been taken with our iPhone from the perspective of looking down at the image on the table. The resulting image is at an angle and rotated. **Bottom:** We have a side-by-side comparison. Our partially completed form has been aligned and registered (*left*) to be more similar to our template form (*right*). Our OpenCV keypoint matching algorithm did a pretty nice job!

An alternative visualization can be seen in Figure 6.6. Here we have overlayed the output aligned image on top of the template.

**Figure 6.6.** This time, we overlay our aligned and registered image on our template with a 50/50 blend. Our overlay method makes the differences pop. Notice there is a slight effect of “double vision” where there are some minor differences.

Our alignment isn't *perfect* (obtaining a pixel-perfect alignment is incredibly challenging and, in some cases, unrealistic). Still, the form's fields are sufficiently aligned such that we'll be able to OCR the text and associate the fields together.

### 6.3 Summary

In this chapter, you learned how to perform image alignment and registration using OpenCV.

Image alignment has many use cases, including medical scans, military-based automatic target acquisition, and satellite image analysis.

We chose to examine image alignment for one of the most important (and most utilized) purposes — **optical character recognition (OCR)**.

Applying image alignment to an input image allows us to align it with a template document. Once we have the input image aligned with the template, we can apply OCR to recognize the text in each field. And since we know the location of each of the document fields, it becomes easy to associate the OCR'd text with each field.

## Chapter 7

# OCR'ing a Document, Form, or Invoice

In this chapter, you will learn how to OCR a document, form, or invoice using Tesseract, OpenCV, and Python.

Previously, we discovered how to accept an input image and align it to a template image. Recall that we had a template image — in our case, it was a form from the U.S. Internal Revenue Service. We also had the image that we wished to align with the template, allowing us to match fields. The process resulted in the alignment of both the template and the form for OCR.

We just need to associate fields on the form such as name, address, EIN, and more. Knowing where and what the fields are allows us to OCR each field and keep track of them for further processing (e.g., automated database entry). However, that raises the questions:

- How do we go about implementing this document OCR pipeline?
- What OCR algorithms will we need to utilize?
- And how *complicated* is this OCR application going to be?
- And most importantly, can we exercise our computer vision and OCR skills to implement our entire document OCR pipeline in under 150 lines of code?

Of course, I wouldn't ask such a question unless the answer is "yes." Here's why we are spending so much time OCR'ing forms. Forms are everywhere.

Schools have many forms for application and enrollment. When's the last time you've gone through the scholarship process?

Healthcare is also full of forms. Both the doctor and the dentist are likely to ask you to fill out a form at your next visit.

We also get forms for property like real estate and homeowners insurance and don't forget about business forms.

If you want to vote, get a permit, or get a passport, you are in for more forms.

As you can see, forms are everywhere so we want to learn to OCR them, let's dive in!

## 7.1 Automatically Registering and OCR'ing a Form

In the first part of this chapter, we'll briefly discuss why we may want to OCR documents, forms, invoices, or any type of physical document. From there, we'll review the steps required to implement a document OCR pipeline. We'll then implement each of the individual steps in a Python script using OpenCV and Tesseract. Finally, we'll review the results of applying image alignment and OCR to our sample images.

### 7.1.1 Why Use OCR on Forms, Invoices, and Documents?

Despite living in the digital age, we still have a strong reliance on *physical* paper trails, especially in large organizations such as government, enterprise companies, and universities/colleges. The need for physical paper trails combined with the fact that nearly every document needs to be organized, categorized, and even *shared* with multiple people in an organization requires that we also *digitize* the information on the document and store it in our databases. Large organizations employ data entry teams whose sole purpose is to take these physical documents, manually re-type the information, and then save it into the system.

Optical character recognition (OCR) algorithms can *automatically* digitize these documents, extract the information, and pipe them into a database for storage, alleviating the need for large, expensive, and even error-prone manual entry teams. In this chapter, you'll learn how to implement a basic document OCR pipeline using OpenCV and Tesseract.

### 7.1.2 Steps to Implementing a Document OCR Pipeline with OpenCV and Tesseract

Implementing a document OCR pipeline with OpenCV and Tesseract is a multi-step process. In this section, we'll discover the **five steps** required for creating a pipeline to OCR a form.

**Step #1** involves defining the locations of fields in the input image document. We can do this by opening our template image in our favorite image editing software, such as Photoshop, GNU Image Manipulation Program (GIMP), or whatever photo application is built into your operating system. From there, we manually examine the image and determine the bounding box ( $x, y$ )-coordinates of each field we want to OCR, as shown on the *top* of Figure 7.1.

**Employee's Withholding Certificate**

► Complete Form W-4 so that your employer can withhold the correct federal income tax from your pay.

► Give Form W-4 to your employer.

► Your withholding is subject to review by the IRS.

**Step 1:** Enter Personal Information

(a) First name and middle initial	Last name
Address	
City or town, state, and ZIP code	
(c) <input type="checkbox"/> Single or Married filing separately <input type="checkbox"/> Married filing jointly (or Qualifying widow(er)) <input type="checkbox"/> Head of household (Check only if you're unmarried and pay more than half the costs of keeping up a home for yours)	

**Step 2:** Multiple Jobs or Spouse Works

Complete this step if you (1) hold more than one job at a time, or (2) are married filing jointly and your spouse also works. The correct amount of withholding depends on income earned from all of these jobs.  
**Do only one of the following.**

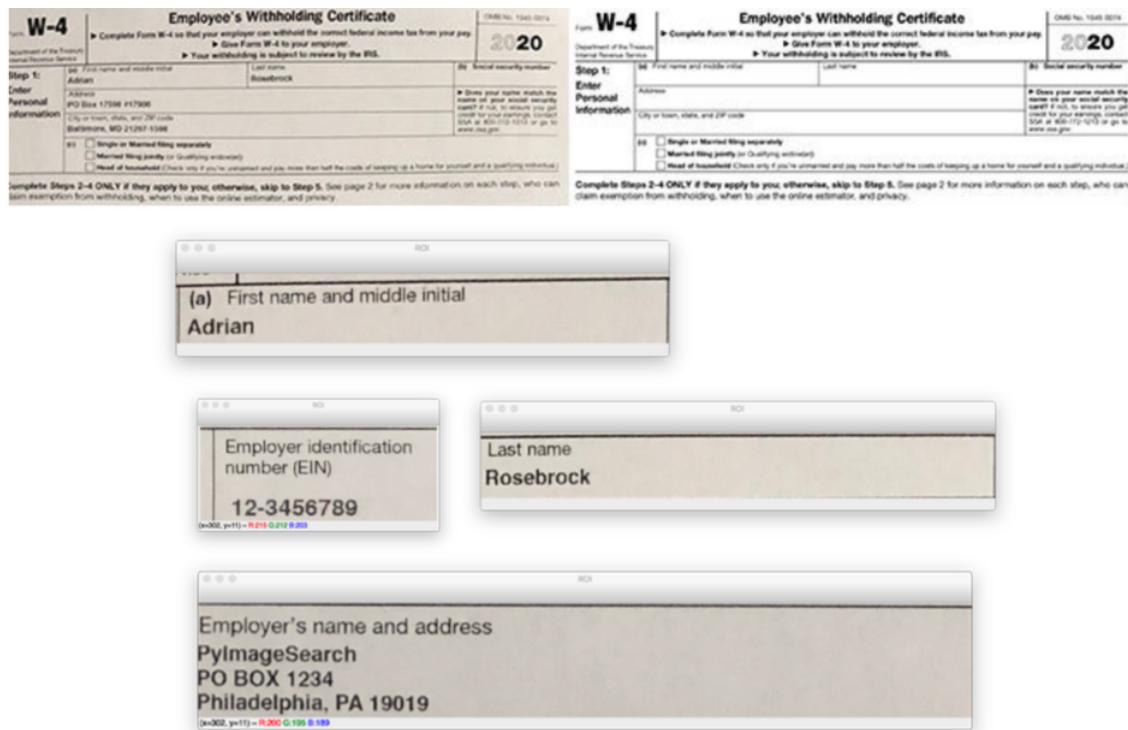
(a) Use the estimator at [www.irs.gov/W4App](http://www.irs.gov/W4App) for most accurate withholding for this step (and Steps 3-4); or  
(b) Use the Multiple Jobs Worksheet on page 3 and enter the result in Step 4(c) below for roughly accurate withholding; or  
(c) If there are only two jobs total, you may check this box. Do the same on Form W-4 for the other job. This option is accurate for jobs with similar pay; otherwise, more tax than necessary may be withheld.

**TIP:** To be accurate, use the estimator at [www.irs.gov/W4App](http://www.irs.gov/W4App).

**Figure 7.1.** Top: Specifying the locations in a document (i.e., form fields) is **Step #1** in implementing a document OCR pipeline with OpenCV, Tesseract, and Python. Bottom: Presenting an image (e.g., a document scan or smartphone photo of a document on a desk) to our OCR pipeline is **Step #2** in our automated OCR system based on OpenCV, Tesseract, and Python.

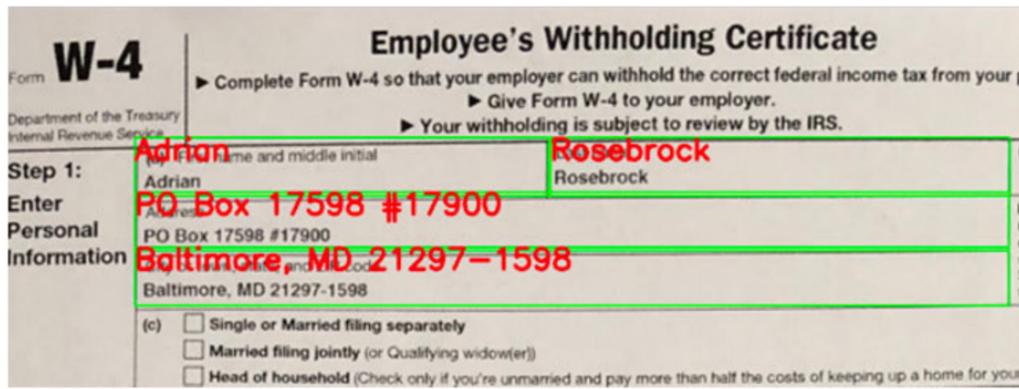
Then, in **Step #2**, we accept an input image containing the document we want to OCR and present it to our OCR pipeline. Figure 7.1 shows both the image *top-left* and the original form *top-right*.

We can then (**Step #3**) apply automatic image alignment/registration to align the input image with the template form (Figure 7.2, *top-left*). In **Step #4**, we loop over all text field locations (which we defined in **Step #1**), extract the ROI, and apply OCR to the ROI. During this step, we're able to OCR the text itself *and* associate it with a text field in the original template document.



**Figure 7.2.** Top-left: Aligning a scanned document with its template using OpenCV and Python represents **Step #3** of our OCR pipeline. Top-right, Middle, and Bottom: Knowing the form field locations from **Step #1** allows us to perform **Step #4**, which consists of extracting ROIs from our aligned document and accomplishing OCR.

The final **Step #5** is to display our output OCR'd document depicted in Figure 7.3.



**Figure 7.3.** Finally, **Step #5** in our OCR pipeline takes action with the OCR'd text data. Given that this is a proof of concept, we'll simply annotate the OCR'd text data on the aligned scan for verification. This is the point where a real-world system would pipe the information into a database or make a decision based upon it (e.g., perhaps you need to apply a mathematical formula to several fields in your document).

We'll learn how to develop a Python script to accomplish **Steps #1–#5** in this chapter by creating an OCR document pipeline using OpenCV and Tesseract.

### 7.1.3 Project Structure

---

```
|-- pyimagesearch
|   |-- alignment
|   |   |-- __init__.py
|   |   |-- align_images.py
|   |-- __init__.py
|   |-- helpers.py
|-- scans
|   |-- scan_01.jpg
|   |-- scan_02.jpg
|-- form_w4.png
|-- ocr_form.py
```

---

The directory and file structure for this chapter are very straightforward. We have three images:

- `scans/scan_01.jpg`: A sample IRS W-4 has been filled with fake tax information, including my name.
- `scans/scan_02.jpg`: A similar sample IRS W-4 document that has been populated with fake tax information.
- `form_w4.png`: The official 2020 IRS W-4 form template. This empty form does not have any information entered into it. We need it and the field locations to line up the scans and ultimately extract information from the scans. We'll manually determine the field locations with an external photo editing/previewing application.

And we have just a single Python driver script to review: `ocr_form.py`. This form parser relies on two helper functions in the `pyimagesearch` module that we're familiar with from previous chapters:

- `align_images`: Contained within the `alignment` submodule
- `cleanup_text`: In our `helpers.py` file

If you're ready to dive in, simply head to the implementation section next!

### 7.1.4 Implementing Our Document OCR Script with OpenCV and Tesseract

We are now ready to implement our document OCR Python script using OpenCV and Tesseract. Open a new file, name it `ocr_form.py`, and insert the following code:

---

```

1 # import the necessary packages
2 from pyimagesearch.alignment import align_images
3 from pyimagesearch.helpers import cleanup_text
4 from collections import namedtuple
5 import pytesseract
6 import argparse
7 import imutils
8 import cv2

```

---

You should recognize each of the imports on **Lines 2–8**; however, let's highlight a few of them. The `cleanup_text` helper function is used to strip out non-ASCII text from a string. We need to cleanse our text because OpenCV's `cv2.putText` is unable to draw non-ASCII characters on an image (unfortunately, OpenCV replaces each unknown character with a `?`). Of course, our effort is a lot easier when we use OpenCV, PyTesseract, and `imutils`.

Next, let's handle our command line arguments:

---

```

10 # construct the argument parser and parse the arguments
11 ap = argparse.ArgumentParser()
12 ap.add_argument("-i", "--image", required=True,
13     help="path to input image that we'll align to template")
14 ap.add_argument("-t", "--template", required=True,
15     help="path to input template image")
16 args = vars(ap.parse_args())

```

---

Our script requires two command line arguments: (1) `--image` is our input image of a form or invoice and (2) `--template` is the path to our template form or invoice. We'll align our image to the template and then OCR various fields as needed.

We aren't creating a "smart form OCR system" in which all text is recognized and fields are designed based on regular expression patterns. That is certainly doable, but to keep this chapter lightweight, I've manually defined `OCR_Locations` for each field about which we are concerned. The benefit is that we'll be able to give each field a name and specify the exact  $(x, y)$ -coordinates serving as the bounds of the field. Let's work on defining the text field locations in **Step #1** now:

---

```

18 # create a named tuple which we can use to create locations of the
19 # input document which we wish to OCR
20 OCRLocation = namedtuple("OCRLocation", ["id", "bbox",
21     "filter_keywords"])
22
23 # define the locations of each area of the document we wish to OCR
24 OCR_LOCATIONS = [
25     OCRLocation("step1_first_name", (265, 237, 751, 106),
26         ["middle", "initial", "first", "name"]),

```

---

```

27     OCRLocation("step1_last_name", (1020, 237, 835, 106),
28         ["last", "name"]),
29     OCRLocation("step1_address", (265, 336, 1588, 106),
30         ["address"]),
31     OCRLocation("step1_city_state_zip", (265, 436, 1588, 106),
32         ["city", "zip", "town", "state"]),
33     OCRLocation("step5_employee_signature", (319, 2516, 1487, 156),
34         ["employee", "signature", "form", "valid", "unless",
35             "you", "sign"]),
36     OCRLocation("step5_date", (1804, 2516, 504, 156), ["date"]),
37     OCRLocation("employee_name_address", (265, 2706, 1224, 180),
38         ["employer", "name", "address"]),
39     OCRLocation("employee_ein", (1831, 2706, 448, 180),
40         ["employer", "identification", "number", "ein"]),
41 ]

```

---

Here, **Lines 20 and 21** create a named tuple consisting of the following:

- "OCRLocation": The name of our tuple.
- "id": A short description of the field for easy reference. Use this field to describe what the form field is. For example, is it a zip-code field?
- "bbox": The bounding box coordinates of a field in list form using the following order: [x, y, w, h]. In this case, x and y are the *top-left* coordinates, and w and h are the width and height.
- "filter\_keywords": A list of words that we do not wish to consider for OCR, such as form field instructions, as demonstrated in Figure 7.4.

**W-4**

Form W-4  
Employee's Withholding Certificate  
Department of the Treasury  
Internal Revenue Service

► Complete Form W-4 so that your employer can withhold the correct federal income tax from your pay.  
► Give Form W-4 to your employer.  
► Your withholding is subject to review by the IRS.

OMB No. 1545-0074  
**2020**

**Step 1: Enter Personal Information**

(a) First name and middle initial	Last name	(b) Social security number
Address		► Does your name match the name on your social security card? If not, to ensure you get credit for your earnings, contact SSA at 800-772-1213 or go to <a href="http://www.ssa.gov">www.ssa.gov</a> .
City or town, state, and ZIP code		
(c) <input type="checkbox"/> Single or Married filing separately <input type="checkbox"/> Married filing jointly (or Qualifying widow(er)) <input type="checkbox"/> Head of household (Check only if you're unmarried and pay more than half the costs of keeping up a home for yourself and a qualifying individual.)		

**Step 5: Sign Here**

Under penalties of perjury, I declare that this certificate, to the best of my knowledge and belief, is true, correct, and complete.	
► Employee's signature (This form is not valid unless you sign it.)	
Date	

**Employers Only**

Employer's name and address	First date of employment	Employer identification number (EIN)
-----------------------------	--------------------------	--------------------------------------

For Privacy Act and Paperwork Reduction Act Notice, see page 3. Cat. No. 10220Q Form W-4 (2020)

Figure 7.4. A W-4 form with eight fields outlined in red.

**Lines 24–41** define **eight fields of an official 2020 IRS W-4 tax form**, pictured in Figure 7.4. Bounding box coordinates ("bbox") were manually determined by inspecting the (x, y)-coordinates of the image. This can be accomplished using any photo editing application, including Photoshop, GIMP, or the basic preview/paint application built into your operating system. Alternatively, you could use OpenCV mouse click events per my blog post, *Capturing Mouse Click Events with Python and OpenCV* (<http://pyimg.co/fmckl> [24]).

Now that we've handled imports, configured command line arguments, and defined our OCR field locations, let's go ahead and **load and align** our input --image to our --template (**Step #2 and Step #3**):

---

```

43 # load the input image and template from disk
44 print("[INFO] loading images...")
45 image = cv2.imread(args["image"])
46 template = cv2.imread(args["template"])
47
48 # align the images
49 print("[INFO] aligning images...")
50 aligned = align_images(image, template)

```

---

As you can see, **Lines 45 and 46** load both our input --image (e.g., a scan or snap from your smartphone camera) and our --template (e.g., a document straight from the IRS, your mortgage company, accounting department, or anywhere else depending on your needs).

**Remark 7.1.** You may be wondering how I converted the *form\_w4.png* from PDF form (most IRS documents are PDFs these days). This process is straightforward with a free OS-agnostic tool called *ImageMagick* (Figure 7.5). With *ImageMagick* installed, you can simply use the *convert* command (<http://pyimg.co/ekwbu> [25]). For example, you could enter the following command:

---

```
$ convert /path/to/taxes/2020/forms/form_w4.pdf ./form_w4.png
```

---

As you can see, *ImageMagick* is smart enough to recognize that you want to convert a PDF to a PNG image based on a combination of the file extension as well as the file itself. You could alter the command quite easily to produce a JPG if you'd like. Once your form is in PNG or JPG form, you can use it with OpenCV and PyTesseract! Do you have a lot of forms? Simply use the *mogrify* command, which supports wildcard operators (<http://pyimg.co/t16bj> [26]).

Once the image files are loaded into memory, we simply take advantage of our *align\_images* helper utility (**Line 50**) to perform the alignment. Figure 7.6 shows the result of aligning our *scan01.jpg* input to the form template. Notice how our input image (*left*) has been aligned to the template document (*right*).

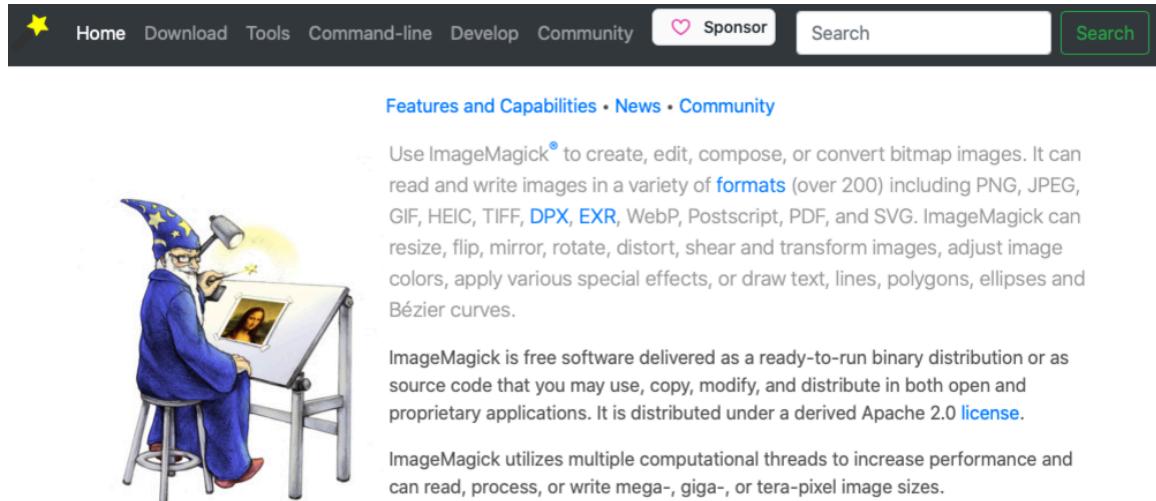


Figure 7.5. ImageMagick software webpage.

The figure displays two versions of the W-4 Employee's Withholding Certificate. The left version is a scanned document with handwritten signatures and some redacted areas. The right version is a digital, fillable form with all fields completed. Both forms include sections for personal information, multiple jobs, dependents, other adjustments, and a signature section at the bottom.

Employee's Withholding Certificate	
Form W-4 Department of the Treasury Internal Revenue Service	
<p>► Complete Form W-4 so that your employer can withhold the correct federal income tax from your pay ► Give Form W-4 to your employer.  <b>2020</b></p> <p>Your withholding is subject to review by the IRS.</p> <p><b>Step 1: Enter Personal Information</b></p> <p>First name and middle initial: Rosebrock Last name: Social security number:</p> <p>Address: PO Box 17508 R17508 City or town, state, and ZIP code: Baltimore, MD 21297-1598</p> <p><input type="checkbox"/> Single or Married filing separately <input type="checkbox"/> Married filing jointly (or Qualifying widow(er)) <input type="checkbox"/> Head of household (Check only if you are unmarried and pay more than half the costs of keeping up a home for yourself and a qualifying individual.)</p> <p>► Complete Steps 2-4 ONLY if they apply to you; otherwise, skip to Step 5. See page 2 for more information on each step, who can claim exemption from withholding, when to use the online estimator, and privacy.</p> <p><b>Step 2: Multiple Jobs or Spouse Works</b></p> <p>Complete this step if you (1) hold more than one job at a time, or (2) are married filing jointly and your spouse also works. The correct amount of withholding depends on income earned from all of these jobs.  <input type="checkbox"/> Do only one of the following:      (a) Use the estimator at <a href="http://www.irs.gov/W4App">www.irs.gov/W4App</a> for most accurate withholding for this step (and Steps 3-6); or      (b) Use the Multiple Jobs Worksheet on page 3 and enter the result in Step 4(b) below for roughly accurate withholding; or      (c) If there are only two jobs total, you may check this box. Do the same on Form W-4 for the other job. This option is accurate for jobs with similar pay; otherwise, more tax than necessary may be withheld. □</p> <p>TIP: To be accurate, submit a 2020 Form W-4 for all other jobs. If you (or your spouse) have self-employment income, including as an independent contractor, use the estimator.</p> <p>► Complete Steps 3-6(b) on Form W-4 for only ONE of these jobs. Leave those steps blank for the other jobs. (Your withholding will be most accurate if you complete Steps 3-6(b) on the Form W-4 for the highest paying job.)</p> <p><b>Step 3: Claim Dependents</b></p> <p>If your income will be \$200,000 or less (\$400,000 or less if married filing jointly):      Multiply the number of qualifying children under age 17 by \$2,000 ► \$ .      Multiply the number of other dependents by \$500 ► \$ .</p> <p>Add the amounts above and enter the total here: <b>3</b> <b>5</b></p> <p><b>Step 4 (optional): Other Adjustments</b></p> <p>(a) <b>Other income (not from jobs).</b> If you want tax withheld for other income you expect this year that won't have withholding, enter the amount of other income here. This may include interest, dividends, and retirement income. <b>4(a) 5</b>      (b) <b>Deductions.</b> If you expect to claim deductions other than the standard deduction and want to reduce your withholding, use the Deductions Worksheet on page 3 and enter the result here. <b>4(b) 5</b>      (c) <b>Extra withholding.</b> Enter any additional tax you want withheld each pay period. <b>4(c) 5</b></p> <p><b>Step 5: Sign Here</b></p> <p>Under penalties of perjury, I declare that this certificate, to the best of my knowledge and belief, is true, correct, and complete.      Adrian Rosebrock      Employee's signature (This form is not valid unless you sign it.) <b>2020/06/10</b>      Date</p> <p><b>Employer Only</b></p> <p>Employer's name and address: PNC Bank NA PO Box 1234 Philadelphia, PA 19109      First date of employment: <b>12-3456789</b>      Employer identification number (EIN):</p> <p>For Privacy Act and Paperwork Reduction Act Notice, see page 3. Cat. No. 10219G Form W-4 (2020)</p>	

Figure 7.6. Left: an image of a W-4 form. Right: a W-4 Form.

The next step (**Step #4**) is to loop over each of our `OCR_LOCATIONS` and **apply OCR to each of the text fields** using the power of Tesseract and PyTesseract:

```
52 # initialize a results list to store the document OCR parsing results
53 print("[INFO] OCR'ing document...")
```

---

```

54 parsingResults = []
55
56 # loop over the locations of the document we are going to OCR
57 for loc in OCR_LOCATIONS:
58     # extract the OCR ROI from the aligned image
59     (x, y, w, h) = loc.bbox
60     roi = aligned[y:y + h, x:x + w]
61
62     # OCR the ROI using Tesseract
63     rgb = cv2.cvtColor(roi, cv2.COLOR_BGR2RGB)
64     text = pytesseract.image_to_string(rgb)

```

---

First, we initialize the `parsingResults` list to store our OCR results for each field of text (**Line 54**). From there, we proceed to loop over each of the `OCR_LOCATIONS` (beginning on **Line 57**), which we have previously manually defined.

Inside the loop (**Lines 59–64**), we begin by (1) extracting the particular text field ROI from the aligned image and (2) use PyTesseract to OCR the ROI. Remember, Tesseract expects an RGB format image, so **Line 63** swaps color channels accordingly.

Now let's break each OCR'd `text` field into individual lines/rows:

---

```

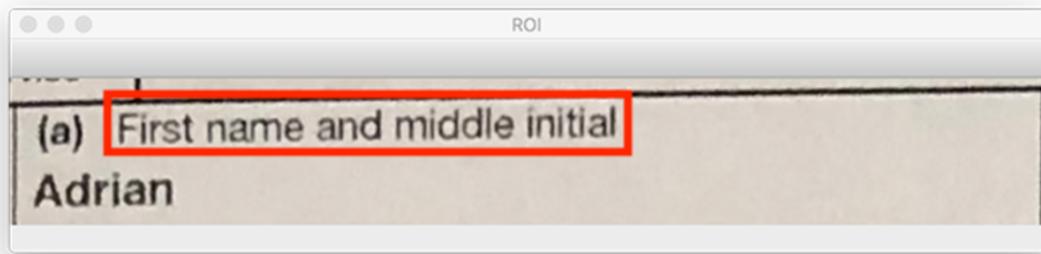
66     # break the text into lines and loop over them
67     for line in text.split("\n"):
68         # if the line is empty, ignore it
69         if len(line) == 0:
70             continue
71
72         # convert the line to lowercase and then check to see if the
73         # line contains any of the filter keywords (these keywords
74         # are part of the *form itself* and should be ignored)
75         lower = line.lower()
76         count = sum([lower.count(x) for x in loc.filter_keywords])
77
78         # if the count is zero then we know we are *not* examining a
79         # text field that is part of the document itself (ex., info,
80         # on the field, an example, help text, etc.)
81         if count == 0:
82             # update our parsing results dictionary with the OCR'd
83             # text if the line is *not* empty
84             parsingResults.append((loc, line))

```

---

**Line 67** begins a loop over the `text` lines where we immediately ignore empty lines (**Lines 69 and 70**). Assuming the `line` isn't empty, we filter it for keywords (forcing to lower-case characters in the process) to ensure that we aren't examining a part of the document itself. In other words, we only care about form-filled information and not the instructional text on the template form itself. **Lines 75–84** accomplish the filtering process and add the OCR'd field to `parsingResults` accordingly.

For example, consider the “*First name and middle initial*” field in Figure 7.7. While I’ve filled out this field with my first name, “Adrian,” the text “(a) *First name and middle initial*” will still be OCR’d by Tesseract — the code above *automatically filters out* the instructional text inside the field, ensuring only the human inputted text is returned.



**Figure 7.7.** First and middle initial field filled in with Adrian

I hope you’re still with me. Let’s carry on by post-processing our `parsingResults` to clean them up:

---

```

86 # initialize a dictionary to store our final OCR results
87 results = {}
88
89 # loop over the results of parsing the document
90 for (loc, line) in parsingResults:
91     # grab any existing OCR result for the current ID of the document
92     r = results.get(loc.id, None)
93
94     # if the result is None, initialize it using the text and location
95     # namedtuple (converting it to a dictionary as namedtuples are not
96     # hashable)
97     if r is None:
98         results[loc.id] = (line, loc._asdict())
99
100    # otherwise, there exists a OCR result for the current area of the
101    # document, so we should append our existing line
102 else:
103     # unpack the existing OCR result and append the line to the
104     # existing text
105     (existingText, loc) = r
106     text = "{}\n{}".format(existingText, line)
107
108     # update our results dictionary
109     results[loc["id"]] = (text, loc)

```

---

Our final `results` dictionary (**Line 87**) will soon hold the cleansed parsing results consisting of the unique ID of the text location (key) and the 2-tuple of the OCR’d text and its location

(value). Let's begin populating our `results` by looping over our `parsingResults` on **Line 90**. Our loop accomplishes three tasks:

- We grab any existing result for the current text field ID
- If there is no current result, we simply store the `text line` and `text loc` (location) in the `results` dictionary
- Otherwise, we *append the line* to any `existingText` separated by a newline for the field and update the `results` dictionary

We're finally ready to perform **Step #5** — visualizing our OCR results:

---

```

111  # loop over the results
112  for (locID, result) in results.items():
113      # unpack the result tuple
114      (text, loc) = result
115
116      # display the OCR result to our terminal
117      print(loc["id"])
118      print("=" * len(loc["id"]))
119      print("{}\n\n".format(text))
120
121      # extract the bounding box coordinates of the OCR location and
122      # then strip out non-ASCII text so we can draw the text on the
123      # output image using OpenCV
124      (x, y, w, h) = loc["bbox"]
125      clean = cleanup_text(text)
126
127      # draw a bounding box around the text
128      cv2.rectangle(aligned, (x, y), (x + w, y + h), (0, 255, 0), 2)
129
130      # loop over all lines in the text
131      for (i, line) in enumerate(text.split("\n")):
132          # draw the line on the output image
133          startY = y + (i * 70) + 40
134          cv2.putText(aligned, line, (x, startY),
135                      cv2.FONT_HERSHEY_SIMPLEX, 1.8, (0, 0, 255), 5)

```

---

Looping over each of our `results` begins on **Line 112**. Our first task is to unpack the 2-tuple consisting of the OCR'd and parsed `text` as well as its `loc` (location) via **Line 114**. Both of these results are then printed in our terminal (**Lines 117–119**).

From there, we extract the bounding box coordinates of the text field (**Line 124**). Subsequently, we strip out non-ASCII characters from the OCR'd `text` via our `cleanup_text` helper utility (**Line 125**). Cleaning up our `text` ensures we can use OpenCV's `cv2.putText` function to annotate the output image.

We then proceed to draw the bounding box rectangle around the `text` (**Line 128**) and annotate each `line` of `text` (delimited by newlines) on the output image (**Lines 131–135**).

Finally, we'll display our (1) original input --image and (2) annotated output result:

---

```
137 # show the input and output images, resizing it such that they fit
138 # on our screen
139 cv2.imshow("Input", imutils.resize(image, width=700))
140 cv2.imshow("Output", imutils.resize(aligned, width=700))
141 cv2.waitKey(0)
```

---

As you can see, our `cv2.imshow` commands also apply aspect-aware resizing because high resolution scans tend not to fit on the average computer screen (**Lines 139 and 140**). To stop the program, simply press any key while one of the windows is in focus.

Great job implementing your automated OCR system with Python, OpenCV, and Tesseract! In the next section, we'll put it to the test.

### 7.1.5 OCR Results Using OpenCV and Tesseract

We are now ready to OCR our document using OpenCV and Tesseract. Fire up your terminal, find the chapter directory, and enter the following command:

---

```
$ python ocr_form.py --image scans/scan_01.jpg --template form_w4.png
[INFO] loading images...
[INFO] aligning images...
[INFO] OCR'ing document...
step1_first_name
=====
Adrian

step1_last_name
=====
Rosebrock

step1_address
=====
PO Box 17598 #17900

step1_city_state_zip
=====
Baltimore, MD 21297-1598

step5_employee_signature
=====
Adrian Rosebrock
```

step5\_date  
=====  
2020/06/10

employee\_name\_address  
=====  
PylmageSearch  
PO BOX 1234  
Philadelphia, PA 19019

employee\_ein  
=====  
12-3456789

As Figure 7.8 demonstrates, we have our input image and its corresponding template. And in Figure 7.9, we have the output of the image alignment and document the OCR pipeline. Notice how we've successfully aligned our input image with the document template, localize each of the fields, and then OCR each of the fields.

Figure 7.8. Left: an input image. Right: a corresponding template.

Beware that our pipeline *ignores* any line of text inside a field *that is part of the document itself*. For example, the first name field provides the instructional text “(a) First name and middle initial.” However, our OCR pipeline and keyword filtering process can detect that this is

## 7.1. Automatically Registering and OCR'ing a Form

101

part of the document itself (i.e., not something a *human* entered) and then simply ignores it. Overall, we've been able to successfully OCR the document!

The screenshot shows the IRS W-4 Employee's Withholding Certificate form. The form is displayed in a window titled "Output". The fields are highlighted with green boxes, indicating successful OCR recognition. Key data extracted includes:

- Name:** Adrian Rosebrock
- Address:** PO Box 17598 #17900
- City, State, Zip:** Baltimore, MD 21297-1598
- Social Security Number:** 2020
- Employer Information:** PymagoSearch, PO Box 1234, Philadelphia, PA 19019
- Date:** 2020/06/10
- First Date of Employment:** 12-3456789
- Other Income:** Adrian
- Dependents:** 3
- Extra Withholding:** None

**Figure 7.9.** Output of the image alignment and document OCR pipeline.

Let's try another sample image, this time with a slightly different viewing angle:

```
$ python ocr_form.py --image scans/scan_02.jpg --template form_w4.png
[INFO] loading images...
[INFO] aligning images...
[INFO] OCR'ing document...
step1_first_name
=====
Adrian

step1_last_name
=====
Rosebrock

step1_address
=====
PO Box 17598 #17900

step1_city_state_zip
=====
Baltimore, MD 21297-1598
```

step5\_employee\_signature  
=====

Adrian Rosebrock

step5\_date  
=====

2020/06/10

employee\_name\_address  
=====

PyimageSearch  
PO BOX 1234  
Philadelphia, PA 19019

employee\_ein  
=====

12-3456789

Figure 7.10 shows our input image along with its template. Figure 7.11 contains our output, where you can see that the image is aligned with the template and the OCR successfully applied to each of the fields. As you can see, we've successfully aligned the input image with the template document and then OCR each of the individual fields!

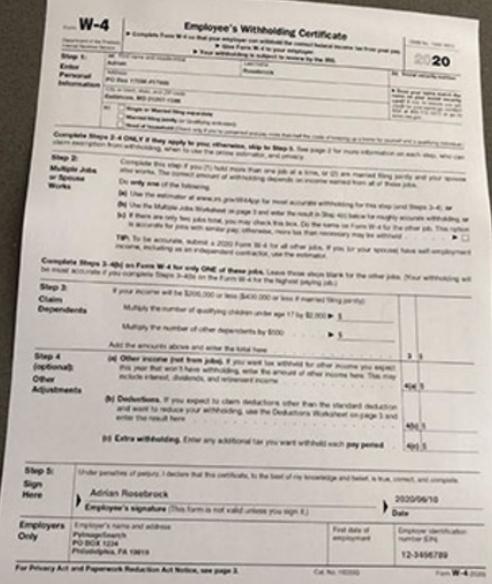
	<table border="0" style="width: 100%; border-collapse: collapse;"> <tr> <td style="text-align: center; padding-bottom: 5px;"> <b>Form W-4</b>  <b>Employee's Withholding Certificate</b>  <small>Department of the Treasury Internal Revenue Service</small> </td> <td style="text-align: right; padding-bottom: 5px;"> <b>2020</b>  <small>GIRS No. 1545-0074</small> </td> </tr> <tr> <td colspan="2" style="text-align: center; padding-top: 5px;"> <b>Employee's Withholding Certificate</b> </td> </tr> <tr> <td colspan="2" style="text-align: center; font-size: small;"> <small>► Complete Form W-4 so that your employer can withhold the correct federal income tax from your pay.</small>  <small>► Give Form W-4 to your employer.</small>  <small>► Your withholding is subject to review by the IRS.</small> </td> </tr> <tr> <td colspan="2" style="text-align: center; font-size: small;"> <small>► Does your name match the names on your social security card? If not, ensure you get corrected information from your Social Security office at 800-772-1273 or go to <a href="http://www.ssa.gov">www.ssa.gov</a>.</small> </td> </tr> <tr> <td colspan="2" style="text-align: center; font-size: small;"> <small>► Social security number</small> </td> </tr> <tr> <td colspan="2" style="text-align: center; font-size: small;"> <small>► First name and middle initial      Last name</small> </td> </tr> <tr> <td colspan="2" style="text-align: center; font-size: small;"> <small>Address</small> </td> </tr> <tr> <td colspan="2" style="text-align: center; font-size: small;"> <small>City or town, state, and ZIP code</small> </td> </tr> <tr> <td colspan="2" style="text-align: center; font-size: small;"> <small>(a) Single or Married filing separately  <input type="checkbox"/>      <input type="checkbox"/> Married filing jointly (or qualifying widow)  <input type="checkbox"/> Head of household (Check only if you're unmarried and pay more than half the costs of keeping up a home for yourself and a qualifying individual.)</small> </td> </tr> <tr> <td colspan="2" style="text-align: center; font-size: small;"> <small>► Complete Steps 2-4 ONLY if they apply to you; otherwise, skip to Step 5. See page 2 for more information on each step, who can claim exemption from withholding, when to use the online estimator, and privacy.</small> </td> </tr> <tr> <td colspan="2" style="text-align: center; font-size: small;"> <small>► Complete this step if you (1) hold more than one job at a time, or (2) are married filing jointly and your spouse also works. The correct amount of withholding depends on income earned from all of these jobs.            Do only one of the following:            (a) Use the estimator at <a href="http://www.irs.gov/W4App">www.irs.gov/W4App</a> for most accurate withholding for this step (and Steps 3-6); or            (b) Use the Multiple Jobs Worksheet on page 3 and enter the result in Step 4(c) below for roughly accurate withholding; or            (c) If there are only two jobs total, you may check this box. Do the same on Form W-4 for the other job. This option is accurate for jobs with similar pay; otherwise, more tax than necessary may be withheld. ►</small> </td> </tr> <tr> <td colspan="2" style="text-align: center; font-size: small;"> <small>TIP: To be accurate, submit a 2020 Form W-4 for all other jobs. If you (or your spouse) have self-employment income, including as an independent contractor, use the estimator.</small> </td> </tr> <tr> <td colspan="2" style="text-align: center; font-size: small;"> <small>► Complete Steps 3-4(b) on Forms W-4 for only ONE of these jobs. Leave those steps blank for the other jobs. (Your withholding will be most accurate if you complete Steps 3-4(b) on the Form W-4 for the highest paying job.)</small> </td> </tr> <tr> <td colspan="2" style="text-align: center; font-size: small;"> <small>► Complete Steps 3-4(b) on Forms W-4 for only ONE of these jobs. Leave those steps blank for the other jobs. (Your withholding will be \$200,000 or less (\$400,000 or less if married filing jointly):            (a) Other income (not from job). If you want tax withheld for other income you expect this year that won't have withholding, enter the amount of other income here. This may include interest, dividends, and retirement income.      4(a) \$ _____            (b) Deductions. If you expect to claim deductions other than the standard deduction and want to reduce your withholding, use the Deductions Worksheet on page 3 and enter the result here.      4(b) \$ _____            (c) Extra withholding. Enter any additional tax you want withheld each pay period.      4(c) \$ _____</small> </td> </tr> <tr> <td colspan="2" style="text-align: center; font-size: small;"> <small>► Add the amounts above and enter the total here.      3 \$ _____</small> </td> </tr> <tr> <td colspan="2" style="text-align: center; font-size: small;"> <small>► Under penalties of perjury, I declare that this certificate, to the best of my knowledge and belief, is true, correct, and complete.</small> </td> </tr> <tr> <td colspan="2" style="text-align: center; font-size: small;"> <small>► Employee's signature (This form is not valid unless you sign it.)      Date</small> </td> </tr> <tr> <td colspan="2" style="text-align: center; font-size: small;"> <small>Employer's name and address PyImageSearch PO BOX 1234 Philadelphia, PA 19019</small> </td> </tr> <tr> <td colspan="2" style="text-align: center; font-size: small;"> <small>First date of employment 2020/06/10</small> </td> </tr> <tr> <td colspan="2" style="text-align: center; font-size: small;"> <small>Employee identification number (EIN) 12-3456789</small> </td> </tr> <tr> <td colspan="2" style="text-align: center; font-size: small;"> <small>For Privacy Act and Paperwork Reduction Act Notice, see page 3.</small> </td> </tr> <tr> <td colspan="2" style="text-align: center; font-size: small;"> <small>Form W-4 (2020)</small> </td> </tr> </table>	<b>Form W-4</b> <b>Employee's Withholding Certificate</b> <small>Department of the Treasury Internal Revenue Service</small>	<b>2020</b> <small>GIRS No. 1545-0074</small>	<b>Employee's Withholding Certificate</b>		<small>► Complete Form W-4 so that your employer can withhold the correct federal income tax from your pay.</small> <small>► Give Form W-4 to your employer.</small> <small>► Your withholding is subject to review by the IRS.</small>		<small>► Does your name match the names on your social security card? If not, ensure you get corrected information from your Social Security office at 800-772-1273 or go to <a href="http://www.ssa.gov">www.ssa.gov</a>.</small>		<small>► Social security number</small>		<small>► First name and middle initial      Last name</small>		<small>Address</small>		<small>City or town, state, and ZIP code</small>		<small>(a) Single or Married filing separately  <input type="checkbox"/>      <input type="checkbox"/> Married filing jointly (or qualifying widow)  <input type="checkbox"/> Head of household (Check only if you're unmarried and pay more than half the costs of keeping up a home for yourself and a qualifying individual.)</small>		<small>► Complete Steps 2-4 ONLY if they apply to you; otherwise, skip to Step 5. See page 2 for more information on each step, who can claim exemption from withholding, when to use the online estimator, and privacy.</small>		<small>► Complete this step if you (1) hold more than one job at a time, or (2) are married filing jointly and your spouse also works. The correct amount of withholding depends on income earned from all of these jobs.            Do only one of the following:            (a) Use the estimator at <a href="http://www.irs.gov/W4App">www.irs.gov/W4App</a> for most accurate withholding for this step (and Steps 3-6); or            (b) Use the Multiple Jobs Worksheet on page 3 and enter the result in Step 4(c) below for roughly accurate withholding; or            (c) If there are only two jobs total, you may check this box. Do the same on Form W-4 for the other job. This option is accurate for jobs with similar pay; otherwise, more tax than necessary may be withheld. ►</small>		<small>TIP: To be accurate, submit a 2020 Form W-4 for all other jobs. If you (or your spouse) have self-employment income, including as an independent contractor, use the estimator.</small>		<small>► Complete Steps 3-4(b) on Forms W-4 for only ONE of these jobs. Leave those steps blank for the other jobs. (Your withholding will be most accurate if you complete Steps 3-4(b) on the Form W-4 for the highest paying job.)</small>		<small>► Complete Steps 3-4(b) on Forms W-4 for only ONE of these jobs. Leave those steps blank for the other jobs. (Your withholding will be \$200,000 or less (\$400,000 or less if married filing jointly):            (a) Other income (not from job). If you want tax withheld for other income you expect this year that won't have withholding, enter the amount of other income here. This may include interest, dividends, and retirement income.      4(a) \$ _____            (b) Deductions. If you expect to claim deductions other than the standard deduction and want to reduce your withholding, use the Deductions Worksheet on page 3 and enter the result here.      4(b) \$ _____            (c) Extra withholding. Enter any additional tax you want withheld each pay period.      4(c) \$ _____</small>		<small>► Add the amounts above and enter the total here.      3 \$ _____</small>		<small>► Under penalties of perjury, I declare that this certificate, to the best of my knowledge and belief, is true, correct, and complete.</small>		<small>► Employee's signature (This form is not valid unless you sign it.)      Date</small>		<small>Employer's name and address PyImageSearch PO BOX 1234 Philadelphia, PA 19019</small>		<small>First date of employment 2020/06/10</small>		<small>Employee identification number (EIN) 12-3456789</small>		<small>For Privacy Act and Paperwork Reduction Act Notice, see page 3.</small>		<small>Form W-4 (2020)</small>	
<b>Form W-4</b> <b>Employee's Withholding Certificate</b> <small>Department of the Treasury Internal Revenue Service</small>	<b>2020</b> <small>GIRS No. 1545-0074</small>																																												
<b>Employee's Withholding Certificate</b>																																													
<small>► Complete Form W-4 so that your employer can withhold the correct federal income tax from your pay.</small> <small>► Give Form W-4 to your employer.</small> <small>► Your withholding is subject to review by the IRS.</small>																																													
<small>► Does your name match the names on your social security card? If not, ensure you get corrected information from your Social Security office at 800-772-1273 or go to <a href="http://www.ssa.gov">www.ssa.gov</a>.</small>																																													
<small>► Social security number</small>																																													
<small>► First name and middle initial      Last name</small>																																													
<small>Address</small>																																													
<small>City or town, state, and ZIP code</small>																																													
<small>(a) Single or Married filing separately  <input type="checkbox"/>      <input type="checkbox"/> Married filing jointly (or qualifying widow)  <input type="checkbox"/> Head of household (Check only if you're unmarried and pay more than half the costs of keeping up a home for yourself and a qualifying individual.)</small>																																													
<small>► Complete Steps 2-4 ONLY if they apply to you; otherwise, skip to Step 5. See page 2 for more information on each step, who can claim exemption from withholding, when to use the online estimator, and privacy.</small>																																													
<small>► Complete this step if you (1) hold more than one job at a time, or (2) are married filing jointly and your spouse also works. The correct amount of withholding depends on income earned from all of these jobs.            Do only one of the following:            (a) Use the estimator at <a href="http://www.irs.gov/W4App">www.irs.gov/W4App</a> for most accurate withholding for this step (and Steps 3-6); or            (b) Use the Multiple Jobs Worksheet on page 3 and enter the result in Step 4(c) below for roughly accurate withholding; or            (c) If there are only two jobs total, you may check this box. Do the same on Form W-4 for the other job. This option is accurate for jobs with similar pay; otherwise, more tax than necessary may be withheld. ►</small>																																													
<small>TIP: To be accurate, submit a 2020 Form W-4 for all other jobs. If you (or your spouse) have self-employment income, including as an independent contractor, use the estimator.</small>																																													
<small>► Complete Steps 3-4(b) on Forms W-4 for only ONE of these jobs. Leave those steps blank for the other jobs. (Your withholding will be most accurate if you complete Steps 3-4(b) on the Form W-4 for the highest paying job.)</small>																																													
<small>► Complete Steps 3-4(b) on Forms W-4 for only ONE of these jobs. Leave those steps blank for the other jobs. (Your withholding will be \$200,000 or less (\$400,000 or less if married filing jointly):            (a) Other income (not from job). If you want tax withheld for other income you expect this year that won't have withholding, enter the amount of other income here. This may include interest, dividends, and retirement income.      4(a) \$ _____            (b) Deductions. If you expect to claim deductions other than the standard deduction and want to reduce your withholding, use the Deductions Worksheet on page 3 and enter the result here.      4(b) \$ _____            (c) Extra withholding. Enter any additional tax you want withheld each pay period.      4(c) \$ _____</small>																																													
<small>► Add the amounts above and enter the total here.      3 \$ _____</small>																																													
<small>► Under penalties of perjury, I declare that this certificate, to the best of my knowledge and belief, is true, correct, and complete.</small>																																													
<small>► Employee's signature (This form is not valid unless you sign it.)      Date</small>																																													
<small>Employer's name and address PyImageSearch PO BOX 1234 Philadelphia, PA 19019</small>																																													
<small>First date of employment 2020/06/10</small>																																													
<small>Employee identification number (EIN) 12-3456789</small>																																													
<small>For Privacy Act and Paperwork Reduction Act Notice, see page 3.</small>																																													
<small>Form W-4 (2020)</small>																																													

Figure 7.10. Left: an input image. Right: a corresponding template.

**Figure 7.11.** Output of the image alignment and document OCR pipeline.

## 7.2 Summary

In this chapter, you learned how to OCR a document, form, or invoice using OpenCV and Tesseract.

Our method hinges on image alignment which we covered earlier in this book, which is the process of accepting an input image and a template image, and then aligning them such that they can neatly “overlap” on top of each other. Image alignment allows us to align each of the text fields in a template with our input image, meaning that once we’ve OCR’d the document, we can associate the OCR’d text to each field (e.g., name, address, etc.).

Once image alignment was applied, we used Tesseract to localize text in the input image. Tesseract’s text localization procedure gave us the  $(x, y)$ -coordinates of each text location, which we then mapped to the appropriate text field in the template. In this manner, we scanned and recognized each important text field of the input document!



## Chapter 8

# Sudoku Solver and OCR

In this chapter, you will create an automatic sudoku puzzle solver using OpenCV, deep learning, and optical character recognition (OCR).

My wife is a *huge* sudoku nerd. Whether it be a 45-minute flight from Philadelphia to Albany or a 6-hour transcontinental flight to California, *she always has a sudoku puzzle with her every time we travel*. **The funny thing is, she always has a printed puzzle book with her.** She hates the digital versions and refuses to play them. I'm not a big puzzle person myself, but one time, we were sitting on a flight, and I asked:

*"How do you know if you solved the puzzle correctly? Is there a solution sheet in the back of the book? Or do you just do it and hope it's correct?"*

That was a stupid question to ask for two reasons. First, **yes, there is a solution key in the back.** All you need to do is flip to the end of the book, locate the puzzle number, and see the solution. And most importantly, **she doesn't solve a puzzle incorrectly.**

My wife doesn't get mad easily, but let me tell you, I touched a nerve when I innocently and unknowingly insulted her sudoku puzzle-solving skills. She then lectured me for 20 minutes on how she only solves "levels 4 and 5 puzzles," followed by a lesson on the "X-wing" and "Y-wing" techniques to solving sudoku puzzles. I have a PhD in computer science, but all of that went over my head. But for those of you who aren't married to a sudoku grandmaster like I am, it does raise the question:

*"Can OpenCV and OCR be used to solve and check sudoku puzzles?"*

If the sudoku puzzle manufacturers *didn't* have to print the answer key in the back of the book and **instead provided an app for users to check their puzzles**, the printers could either pocket the savings or print additional puzzles at no cost. The sudoku puzzle company makes more money, and the end-users are happy. It seems like a win/win to me.

## 8.1 Chapter Learning Objectives

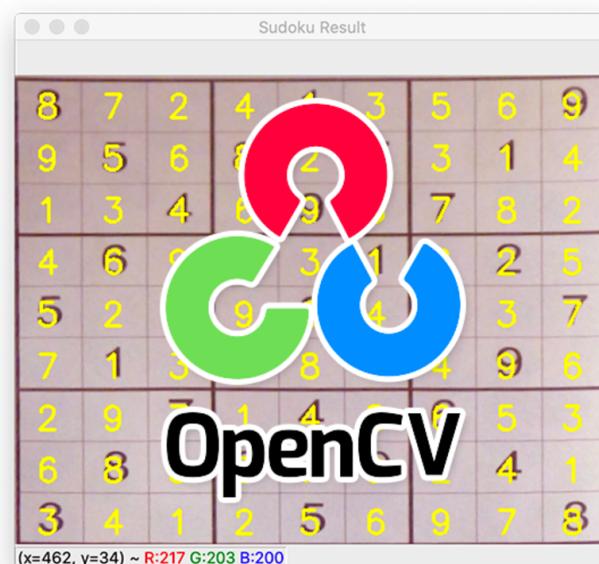
In this chapter, we'll:

- i. Discover the steps required to build a sudoku puzzle solver using OpenCV, deep learning, and OCR techniques
- ii. Install the `py-sudoku` package
- iii. Implement and train a small CNN named `SudokuNet`, which can recognize digits 0–9
- iv. Learn how to pre-process a photo of a sudoku puzzle and ultimately recognize and extract all digits while also identifying the blank spaces. We'll feed the data into `py-sudoku` to solve the puzzle for us.

## 8.2 OpenCV Sudoku Solver and OCR

In the first part of this chapter, we'll discuss the steps required to build a sudoku puzzle solver using OpenCV, deep learning, and OCR techniques. From there, you'll configure your development environment and ensure the proper libraries and packages are installed. Before we write any code, we'll first review our project directory structure, ensuring you know what files will be created, modified, and utilized throughout this chapter.

You can see a solved sudoku puzzle in Figure 8.1.



**Figure 8.1.** In this chapter, we'll use our OpenCV and deep learning skills to extract sudoku puzzle information and send it through a puzzle solver utility. Using a computer, we can solve a sudoku puzzle in less than a second compared to the many tens of minutes you would solve the puzzle by hand.

I'll then show you how to implement `SudokuNet`, a basic convolutional neural network (CNN) used to OCR the digits on the sudoku puzzle board. We'll then train that network to recognize digits using Keras and TensorFlow.

But before we can *check* and *solve* a sudoku puzzle, we first need to locate *where* in the image the sudoku board is — we'll implement helper functions and utilities to help with that task. Finally, we'll put all the pieces together and implement our full OpenCV sudoku puzzle solver.

### 8.2.1 How to Solve Sudoku Puzzles with OpenCV and OCR

Creating an automatic sudoku puzzle solver with OpenCV is a 6-step process:

- **Step #1:** Provide an input image containing a sudoku puzzle to our system.
- **Step #2:** Locate *where* in the input image, the puzzle is and extract the board.
- **Step #3:** Given the board, locate each of the sudoku board's cells (most standard sudoku puzzles are a  $9 \times 9$  grid, so we'll need to localize each of these cells).
- **Step #4:** Determine if a digit exists in the cell, and if so, OCR it.
- **Step #5:** Apply a sudoku puzzle solver/checker algorithm to validate the puzzle.
- **Step #6:** Display the output result to the user.

The majority of these steps can be accomplished using OpenCV and basic computer vision and image processing operations.

**The most significant exception is Step #4, where we need to apply OCR.**

OCR can be a bit tricky to apply, but we have several options:

- i. Use the Tesseract OCR engine, the *de facto* standard for open-source OCR
- ii. Utilize cloud-based OCR APIs (e.g., Microsoft Cognitive Services, Amazon Rekognition API, or the Google Cloud Vision API)
- iii. Train our custom OCR model

All of these are perfectly valid options; however, to make a complete end-to-end chapter, I've decided that we'll train our custom sudoku OCR model using deep learning.

Be sure to strap yourself in — this is going to be a wild ride.

### 8.2.2 Configuring Your Development Environment to Solve Sudoku Puzzles with OpenCV and OCR

This chapter requires that you install the `py-sudoku` package into your environment (<http://pyimg.co/3ipyb> [27]). This library allows us to solve sudoku puzzles (assuming we've already OCR'd all the digits). To install it, simply use pip.

---

```
$ pip install py-sudoku
```

---

### 8.2.3 Project Structure

We'll get started by reviewing our project directory structure:

---

```
|-- output
|   |-- digit_classifier.h5
|-- pyimagesearch
|   |-- models
|       |-- __init__.py
|       |-- sudokunet.py
|   |-- sudoku
|       |-- __init__.py
|       |-- puzzle.py
|   |-- __init__.py
|-- solve_sudoku_puzzle.py
|-- sudoku_puzzle.jpg
|-- train_digit_classifier.py
```

---

Inside, you'll find a `pyimagesearch` module containing the following:

- `sudokunet.py`: Holds the SudokuNet CNN architecture implemented with Keras and TensorFlow.
- `puzzle.py`: Contains two helper utilities for finding the sudoku puzzle board itself as well as digits therein.

As with all CNNs, SudokuNet needs to be trained with data. Our `train_digit_classifier.py` script will train a digit OCR model on the MNIST dataset [1]. Once SudokuNet is successfully trained, we'll deploy it with our `solve_sudoku_puzzle.py` script to solve a sudoku puzzle.

When our system is working, you can impress your friends with the app. Or better yet, fool them on the airplane as you solve puzzles faster than they possibly can in the seat right behind you! Don't worry, I won't tell!

### 8.2.4 SudokuNet: A Digit OCR Model Implemented in Keras and TensorFlow

Every sudoku puzzle starts with an  $N \times N$  grid (typically  $9 \times 9$ ), where **some cells are blank**, and **other cells already contain a digit**. The goal is to use the knowledge about the *existing digits* to *infer the other digits correctly*.

But before we can solve sudoku puzzles with OpenCV, we first need to implement a neural network architecture that will handle OCR'ing the digits on the sudoku puzzle board — given that information. It will become trivial to solve the actual puzzle. Fittingly, we'll name our sudoku puzzle architecture `SudokuNet`. Let's code our `sudokunet.py` file in the `pyimagesearch` module now:

---

```
1 # import the necessary packages
2 from tensorflow.keras.models import Sequential
3 from tensorflow.keras.layers import Conv2D
4 from tensorflow.keras.layers import MaxPooling2D
5 from tensorflow.keras.layers import Activation
6 from tensorflow.keras.layers import Flatten
7 from tensorflow.keras.layers import Dense
8 from tensorflow.keras.layers import Dropout
```

---

All of `SudokuNet`'s imports are from `tf.keras`. As you can see, we'll be using Keras' Sequential API (<http://pyimg.co/mac01> [28]) as well as the `layers` shown.

Now that our imports are taken care of, let's dive right into the implementation of our CNN:

---

```
10 class SudokuNet:
11     @staticmethod
12     def build(width, height, depth, classes):
13         # initialize the model
14         model = Sequential()
15         inputShape = (height, width, depth)
```

---

Our `SudokuNet` class is defined with a single static method (no constructor) on **Lines 10–12**. The `build` method accepts the following parameters:

- `width`: The width of an MNIST digit (28 pixels)
- `height`: The height of an MNIST digit (28 pixels)
- `depth`: Channels of MNIST digit images (1 grayscale channel)
- `classes`: The number of digits 0–9 (10 digits)

**Lines 14 and 15** initialize our `model` to be built with the Sequential API and establish the `inputShape`, which we'll need for our first CNN layer.

Now that our `model` is initialized, let's go ahead and build out our CNN:

---

```

17      # first set of CONV => RELU => POOL layers
18      model.add(Conv2D(32, (5, 5), padding="same",
19                      input_shape=inputShape))
20      model.add(Activation("relu"))
21      model.add(MaxPooling2D(pool_size=(2, 2)))
22
23      # second set of CONV => RELU => POOL layers
24      model.add(Conv2D(32, (3, 3), padding="same"))
25      model.add(Activation("relu"))
26      model.add(MaxPooling2D(pool_size=(2, 2)))
27
28      # first set of FC => RELU layers
29      model.add(Flatten())
30      model.add(Dense(64))
31      model.add(Activation("relu"))
32      model.add(Dropout(0.5))
33
34      # second set of FC => RELU layers
35      model.add(Dense(64))
36      model.add(Activation("relu"))
37      model.add(Dropout(0.5))
38
39      # softmax classifier
40      model.add(Dense(classes))
41      model.add(Activation("softmax"))
42
43      # return the constructed network architecture
44      return model

```

---

The body of our network is composed of:

- CONV => RELU => POOL layer set 1
- CONV => RELU => POOL layer set 2
- FC => RELU fully connected layer set with 50% dropout

The head of the network consists of a softmax classifier with the number of outputs equal to the number of our `classes` (in our case: 10 digits).

Great job implementing SudokuNet!

If the CNN layers and working with the Sequential API were unfamiliar to you, I recommend checking out either of the following resources:

- *Keras Tutorial: How to Get Started with Keras, Deep Learning, and Python* (<http://pyimg.co/8gzi2> [29])
- *Deep Learning for Computer Vision with Python* (Starter Bundle) (<http://pyimg.co/dl4cv> [6])

If you were building a CNN to classify 26 uppercase English letters plus the 10 digits (a total of 36 characters), you most certainly would need a deeper CNN.

### 8.2.5 Implementing with Keras and TensorFlow

With the `SudokuNet` model architecture implemented, we can create a Python script to train the model to recognize digits. Perhaps unsurprisingly, we'll be using the MNIST dataset to train our digit recognizer, as it fits quite nicely in this use case. Open the `train_digit_classifier.py` file to get started:

---

```

1 # import the necessary packages
2 from pyimagesearch.models import SudokuNet
3 from tensorflow.keras.optimizers import Adam
4 from tensorflow.keras.datasets import mnist
5 from sklearn.preprocessing import LabelBinarizer
6 from sklearn.metrics import classification_report
7 import argparse
8
9 # construct the argument parser and parse the arguments
10 ap = argparse.ArgumentParser()
11 ap.add_argument("-m", "--model", required=True,
12     help="path to output model after training")
13 args = vars(ap.parse_args())

```

---

We begin our training script with a small handful of imports. Most notably, we're importing `SudokuNet` and the `mnist` dataset. The MNIST dataset of handwritten digits (<http://pyimg.co/zntI9> [1]) is built right into Keras/TensorFlow's `datasets` module and will be cached to your machine on demand. Figure 8.2 shows a sample of the type of digits used.



**Figure 8.2.** A sample of digits from Yann LeCun's MNIST dataset (<http://pyimg.co/zntI9> [1]) of handwritten digits will be used to train a deep learning model to OCR/HWR handwritten digits with Keras/TensorFlow.

Our script requires a single command line argument. When you execute the training script from the command line, simply pass a filename for your output model file (I recommend using the `.h5` file extension).

Next, we'll (1) set hyperparameters and (2) load and pre-process MNIST:

---

```

15 # initialize the initial learning rate, number of epochs to train
16 # for, and batch size
17 INIT_LR = 1e-3
18 EPOCHS = 10
19 BS = 128
20
21 # grab the MNIST dataset
22 print("[INFO] accessing MNIST...")
23 ((trainData, trainLabels), (testData, testLabels)) = mnist.load_data()
24
25 # add a channel (i.e., grayscale) dimension to the digits
26 trainData = trainData.reshape((trainData.shape[0], 28, 28, 1))
27 testData = testData.reshape((testData.shape[0], 28, 28, 1))
28
29 # scale data to the range of [0, 1]
30 trainData = trainData.astype("float32") / 255.0
31 testData = testData.astype("float32") / 255.0
32
33 # convert the labels from integers to vectors
34 le = LabelBinarizer()
35 trainLabels = le.fit_transform(trainLabels)
36 testLabels = le.transform(testLabels)

```

---

You can configure training hyperparameters on **Lines 17–19**. Through experimentation, I've determined appropriate settings for the learning rate, number of training epochs, and batch size. Advanced users might wish to check out my *Keras Learning Rate Finder* tutorial (<http://pyimg.co/9so5k> [30]).

To work with the MNIST digit dataset, we perform the following steps:

- Load the dataset into memory (**Line 23**). This dataset is already split into training and testing data
- Add a channel dimension to the digits to indicate that they are grayscale (**Lines 26 and 27**)
- Scale data to the range of  $[0, 1]$  (**Lines 30 and 31**)
- One-hot encode labels (**Lines 34–36**)

The process of one-hot encoding means that an integer such as 3 would be represented as follows:

```
[0, 0, 0, 1, 0, 0, 0, 0, 0]
```

Or the integer 9 would be encoded like so:

```
[0, 0, 0, 0, 0, 0, 0, 0, 1]
```

From here, we'll go ahead and initialize and train `SudokuNet` on our digits data:

---

```
38 # initialize the optimizer and model
39 print("[INFO] compiling model...")
40 opt = Adam(lr=INIT_LR)
41 model = SudokuNet.build(width=28, height=28, depth=1, classes=10)
42 model.compile(loss="categorical_crossentropy", optimizer=opt,
43     metrics=[ "accuracy"])
44
45 # train the network
46 print("[INFO] training network...")
47 H = model.fit(
48     trainData, trainLabels,
49     validation_data=(testData, testLabels),
50     batch_size=BS,
51     epochs=EPOCHS,
52     verbose=1)
```

---

**Lines 40–43** build and compile our `model` with the `Adam` optimizer and categorical cross-entropy loss.

**Remark 8.1.** *We're focused on 10 digits. However, if you were only focused on recognizing binary numbers 0 and 1, then you would use `loss="binary_crossentropy"`. Keep this in mind when working with two-class datasets or data subsets.*

Training is launched via a call to the `fit` method (**Lines 47–52**). Once training is complete, we'll evaluate and export our model:

---

```
54 # evaluate the network
55 print("[INFO] evaluating network...")
56 predictions = model.predict(testData)
57 print(classification_report(
58     testLabels.argmax(axis=1),
59     predictions.argmax(axis=1),
60     target_names=[str(x) for x in le.classes_]))
61
62 # serialize the model to disk
63 print("[INFO] serializing digit model...")
64 model.save(args["model"], save_format="h5")
```

---

Using our newly trained `model`, we make predictions on our `testData` (**Line 56**). From there, we print a classification report to our terminal (**Lines 57–60**). Finally, we save our

model to disk (**Line 64**). Note that for TensorFlow 2.0+, we recommend explicitly setting the `save_format="h5"` (HDF5 format).

### 8.2.6 Training Our Sudoku Digit Recognizer with Keras and TensorFlow

We're now ready to train our `SudokuNet` model to recognize digits. Go ahead and open your terminal, and execute the following command:

---

```
$ python train_digit_classifier.py --model output/digit_classifier.h5
[INFO] accessing MNIST...
[INFO] compiling model...
[INFO] training network...
[INFO] training network...
Epoch 1/10
469/469 [=====] - 22s 47ms/step - loss: 0.7311
- accuracy: 0.7530 - val_loss: 0.0989 - val_accuracy: 0.9706
Epoch 2/10
469/469 [=====] - 22s 47ms/step - loss: 0.2742
- accuracy: 0.9168 - val_loss: 0.0595 - val_accuracy: 0.9815
Epoch 3/10
469/469 [=====] - 21s 44ms/step - loss: 0.2083
- accuracy: 0.9372 - val_loss: 0.0452 - val_accuracy: 0.9854
...
Epoch 8/10
469/469 [=====] - 22s 48ms/step - loss: 0.1178
- accuracy: 0.9668 - val_loss: 0.0312 - val_accuracy: 0.9893
Epoch 9/10
469/469 [=====] - 22s 47ms/step - loss: 0.1100
- accuracy: 0.9675 - val_loss: 0.0347 - val_accuracy: 0.9889
Epoch 10/10
469/469 [=====] - 22s 47ms/step - loss: 0.1005
- accuracy: 0.9700 - val_loss: 0.0392 - val_accuracy: 0.9889
[INFO] evaluating network...
      precision    recall   f1-score   support
0          0.98     1.00     0.99      980
1          0.99     1.00     0.99     1135
2          0.99     0.98     0.99     1032
3          0.99     0.99     0.99     1010
4          0.99     0.99     0.99      982
5          0.98     0.99     0.98      892
6          0.99     0.98     0.99      958
7          0.98     1.00     0.99     1028
8          1.00     0.98     0.99      974
9          0.99     0.98     0.99     1009

accuracy                           0.99      10000
macro avg       0.99     0.99     0.99     10000
weighted avg    0.99     0.99     0.99     10000

[INFO] serializing digit model...
```

---

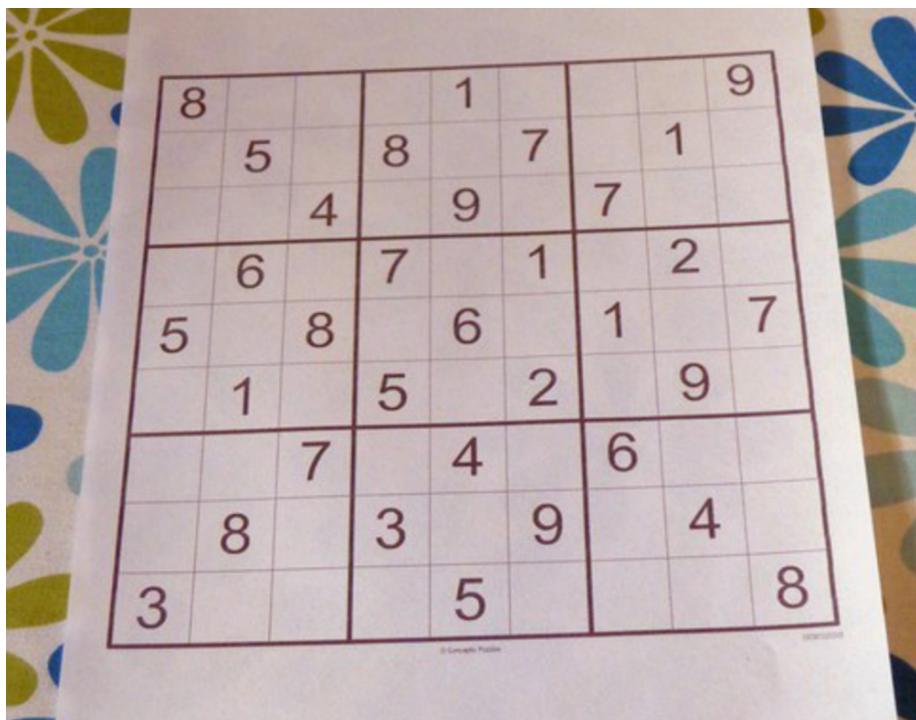
Here, you can see that our `SudokuNet` model has obtained **99% accuracy** on our testing set. You can verify that the model is serialized to disk by inspecting your `output` directory:

```
$ ls -lh output
total 2824
-rw-r--r--@ 1 adrian  staff  1.4M Jun  7 07:38 digit_classifier.h5
```

This `digit_classifier.h5` file contains our Keras/TensorFlow model, which we'll use to recognize the digits on a sudoku board later in this chapter. This model is quite small, making it a good candidate to deploy to low-power devices.

### 8.2.7 Finding the Sudoku Puzzle Board in an Image with OpenCV

At this point, we have a model that can recognize digits in an image; however, **that digit recognizer doesn't do us much good if it can't locate the sudoku puzzle board in an image**. For example, let's say we presented the sudoku puzzle board in Figure 8.3 to our system:



**Figure 8.3.** An example of a sudoku puzzle that has been snapped with a camera (source: Akash Jhawar, <http://pyimg.co/tq5yz> [31]). In this section, we'll develop two computer vision helper utilities with OpenCV and scikit-image. The first helps us find and perform a perspective transform on the puzzle itself. And the second helper allows us to find the digit contours within a cell or determine that a cell is empty.

How are we going to locate the actual sudoku puzzle board in the image? And once we've found the puzzle, how do we identify each of the individual cells?

To make our lives a bit easier, we'll be implementing two helper utilities:

- `find_puzzle`: Locates and extracts the sudoku puzzle board from the input image
- `extract_digit`: Examines each cell of the sudoku puzzle board and extracts the digit from the cell (provided there is a digit)

This section will show you how to implement the `find_puzzle` method, while the next section will show the `extract_digit` implementation. Open the `puzzle.py` file in the `pyimagesearch` module, and we'll get started:

---

```

1 # import the necessary packages
2 from imutils.perspective import four_point_transform
3 from skimage.segmentation import clear_border
4 import numpy as np
5 import imutils
6 import cv2
7
8 def find_puzzle(image, debug=False):
9     # convert the image to grayscale and blur it slightly
10    gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
11    blurred = cv2.GaussianBlur(gray, (7, 7), 3)

```

---

Our two helper functions require my `imutils` (<http://pyimg.co/twcpf> [32]) implementation of a `four_point_transform` for deskewing an image to obtain a bird's-eye view. Additionally, we'll use the `clear_border` routine in our `extract_digit` function to clean up a sudoku cell's edges. Most operations will be driven with OpenCV plus a little bit of help from NumPy and `imutils`. Our `find_puzzle` function comes first and accepts two parameters:

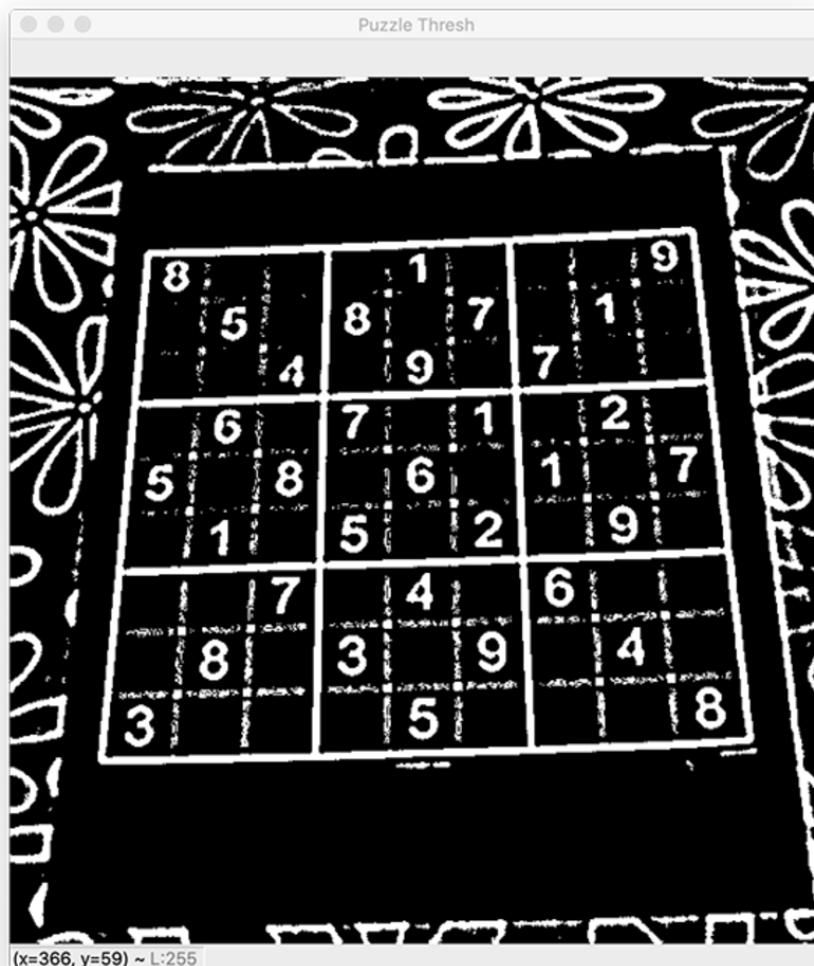
- `image`: The photo of a sudoku puzzle.
- `debug`: An optional Boolean indicating whether to show intermediate steps so you can better visualize what is happening under the hood of our computer vision pipeline. If you encounter any issues, I recommend setting `debug=True` and using your computer vision knowledge to iron out bugs.

Our first step is to convert our `image` to grayscale and apply a Gaussian blur operation with a  $7 \times 7$  kernel (<http://pyimg.co/8om5g> [13]) (**Lines 10 and 11**).

And next, we'll apply adaptive thresholding:

```
13     # apply adaptive thresholding and then invert the threshold map
14     thresh = cv2.adaptiveThreshold(blurred, 255,
15         cv2.ADAPTIVE_THRESH_GAUSSIAN_C, cv2.THRESH_BINARY, 11, 2)
16     thresh = cv2.bitwise_not(thresh)
17
18     # check to see if we are visualizing each step of the image
19     # processing pipeline (in this case, thresholding)
20     if debug:
21         cv2.imshow("Puzzle Thresh", thresh)
22         cv2.waitKey(0)
```

Binary adaptive thresholding operations allow us to peg grayscale pixels toward each end of the  $[0, 255]$  pixel range. In this case, we've both applied a binary threshold and then inverted the result, as shown in Figure 8.4.



**Figure 8.4.** OpenCV has been used to perform a binary inverse threshold operation on the input image.

Just remember, you'll only see something similar to the inverted thresholded image if you have your `debug` option set to `True`. Now that our image is thresholded, let's find and sort contours:

---

```

24     # find contours in the thresholded image and sort them by size in
25     # descending order
26     cnts = cv2.findContours(thresh.copy(), cv2.RETR_EXTERNAL,
27         cv2.CHAIN_APPROX_SIMPLE)
28     cnts = imutils.grab_contours(cnts)
29     cnts = sorted(cnts, key=cv2.contourArea, reverse=True)
30
31     # initialize a contour that corresponds to the puzzle outline
32     puzzleCnt = None
33
34     # loop over the contours
35     for c in cnts:
36         # approximate the contour
37         peri = cv2.arcLength(c, True)
38         approx = cv2.approxPolyDP(c, 0.02 * peri, True)
39
40         # if our approximated contour has four points, then we can
41         # assume we have found the outline of the puzzle
42         if len(approx) == 4:
43             puzzleCnt = approx
44             break

```

---

Here, we find contours and sort by area (<http://pyimg.co/sbm9p> [33]) in reverse order (**Lines 26–29**). One of our contours will correspond to the sudoku grid outline — `puzzleCnt` is initialized to `None` on **Line 32**. Let's determine which of our `cnts` is our `puzzleCnt` using the following approach:

- Loop over all contours beginning on **Line 35**.
- Determine the perimeter of the contour (**Line 37**).
- Approximate the contour (**Line 38**) (<http://pyimg.co/c1tel> [34]).
- Check if the contour has four vertices, and if so, mark it as the `puzzleCnt`, and break out of the loop (**Lines 42–44**).

It is possible that the outline of the sudoku grid isn't found. In that case, let's raise an Exception:

---

```

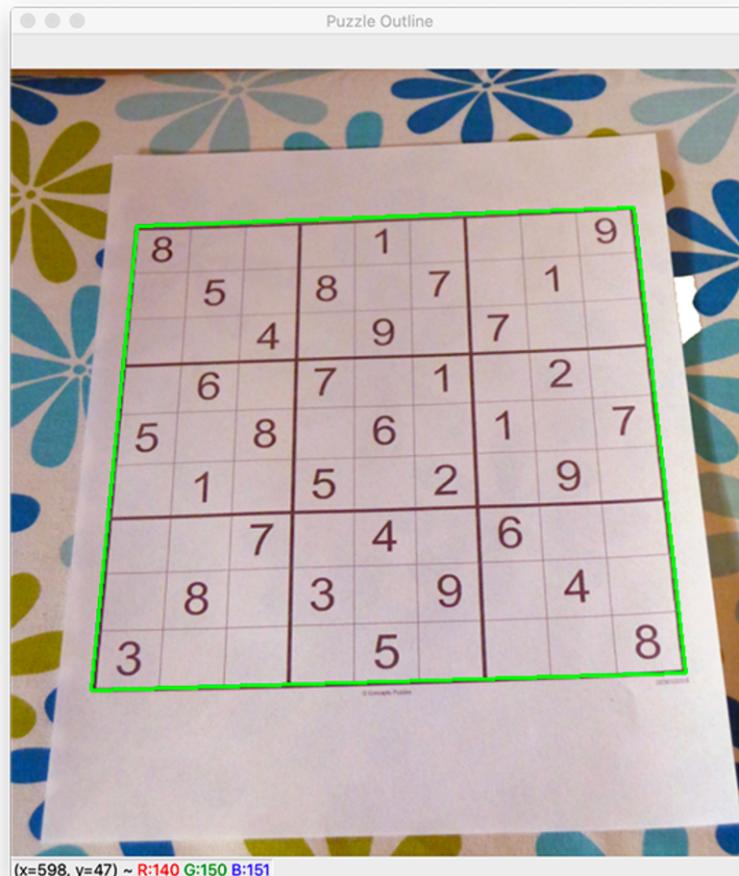
46     # if the puzzle contour is empty then our script could not find
47     # the outline of the sudoku puzzle so raise an error
48     if puzzleCnt is None:

```

---

```
49         raise Exception(("Could not find sudoku puzzle outline. "
50                           "Try debugging your thresholding and contour steps."))
51
52     # check to see if we are visualizing the outline of the detected
53     # sudoku puzzle
54     if debug:
55         # draw the contour of the puzzle on the image and then display
56         # it to our screen for visualization/debugging purposes
57         output = image.copy()
58         cv2.drawContours(output, [puzzleCnt], -1, (0, 255, 0), 2)
59         cv2.imshow("Puzzle Outline", output)
60         cv2.waitKey(0)
```

If the sudoku puzzle is not found, we raise an `Exception` to tell the user/developer what happened (**Lines 48–50**). And again, if we are debugging, we'll visualize what is going on under the hood by drawing the puzzle contour outline on the image, as shown in Figure 8.5:



**Figure 8.5.** The sudoku puzzle board's border is found by determining the largest contour with four points using OpenCV's contour operations.

With the contour of the puzzle in hand (fingers crossed), we're then able to deskew the image to obtain a *top-down* bird's-eye view of the puzzle:

---

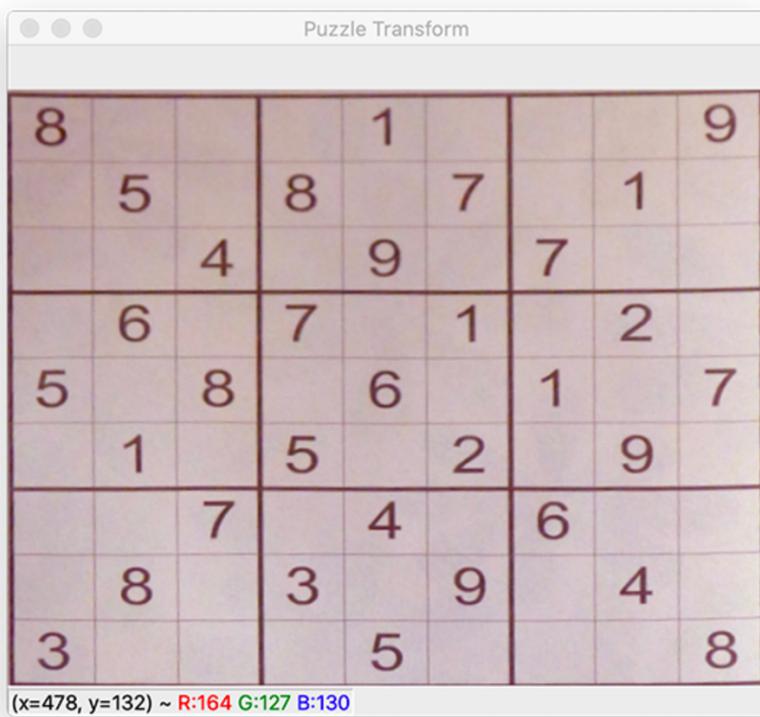
```

62     # apply a four-point perspective transform to both the original
63     # image and grayscale image to obtain a top-down bird's-eye view
64     # of the puzzle
65     puzzle = four_point_transform(image, puzzleCnt.reshape(4, 2))
66     warped = four_point_transform(gray, puzzleCnt.reshape(4, 2))
67
68     # check to see if we are visualizing the perspective transform
69     if debug:
70         # show the output warped image (again, for debugging purposes)
71         cv2.imshow("Puzzle Transform", puzzle)
72         cv2.waitKey(0)
73
74     # return a 2-tuple of puzzle in both RGB and grayscale
75     return (puzzle, warped)

```

---

Applying a four-point perspective transform (<http://pyimg.co/7t1dj> [35]) effectively deskews our sudoku puzzle grid, making it much easier for us to determine rows, columns, and cells as we move forward (**Lines 65 and 66**). This operation is performed on the original RGB `image` and `gray` image. The final result of our `find_puzzle` function is shown in Figure 8.6:



**Figure 8.6.** After applying a four-point perspective transform using OpenCV, we're left with a *top-down* bird's-eye view of the sudoku puzzle. At this point, we can begin working on finding characters and performing deep learning based OCR with Keras/TensorFlow.

Our `find_puzzle` return signature consists of a 2-tuple of the original RGB image and grayscale image after all operations, including the final four-point perspective transform.

Great job so far! Let's continue our forward march toward solving sudoku puzzles. Now we need a means to extract digits from sudoku puzzle cells, and we'll do just that in the next section.

### 8.2.8 Extracting Digits from a Sudoku Puzzle with OpenCV

This section will show you how to examine each of the cells in a sudoku board, detect if there is a digit in the cell, and, if so, extract the digit. Let's open the `puzzle.py` file once again and get to work:

---

```

77 def extract_digit(cell, debug=False):
78     # apply automatic thresholding to the cell and then clear any
79     # connected borders that touch the border of the cell
80     thresh = cv2.threshold(cell, 0, 255,
81                           cv2.THRESH_BINARY_INV | cv2.THRESH_OTSU)[1]
82     thresh = clear_border(thresh)
83
84     # check to see if we are visualizing the cell thresholding step
85     if debug:
86         cv2.imshow("Cell Thresh", thresh)
87         cv2.waitKey(0)
```

---

Here, you can see we've defined our `extract_digit` function to accept two parameters:

- `cell`: An ROI representing an individual cell of the sudoku puzzle (it may or may not contain a digit)
- `debug`: A Boolean indicating whether intermediate step visualizations should be shown on your screen

Our first step, on **Lines 80–82**, is to threshold and clear any foreground pixels that are touching the borders of the cell (e.g., any line markings from the cell dividers). The result of this operation can be shown via **Lines 85–87**. Let's see if we can find the digit contour:

---

```

89     # find contours in the thresholded cell
90     cnts = cv2.findContours(thresh.copy(), cv2.RETR_EXTERNAL,
91                            cv2.CHAIN_APPROX_SIMPLE)
92     cnts = imutils.grab_contours(cnts)
93
94     # if no contours were found than this is an empty cell
95     if len(cnts) == 0:
```

---

```

96     return None
97
98     # otherwise, find the largest contour in the cell and create a
99     # mask for the contour
100    c = max(cnts, key=cv2.contourArea)
101    mask = np.zeros(thresh.shape, dtype="uint8")
102    cv2.drawContours(mask, [c], -1, 255, -1)

```

---

**Lines 90–92** find the contours in the thresholded cell. If no contours are found, we return `None` (**Lines 95 and 96**).

Given our contours, `cnts`, we then find the largest contour by pixel area and construct an associated `mask` (**Lines 100–102**).

From here, we'll continue working on trying to isolate the digit in the cell:

---

```

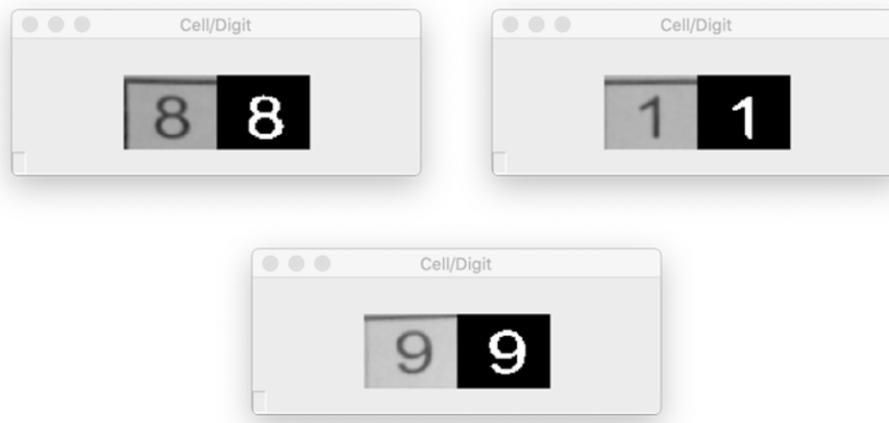
104     # compute the percentage of masked pixels relative to the total
105     # area of the image
106     (h, w) = thresh.shape
107     percentFilled = cv2.countNonZero(mask) / float(w * h)
108
109     # if less than 3% of the mask is filled then we are looking at
110     # noise and can safely ignore the contour
111     if percentFilled < 0.03:
112         return None
113
114     # apply the mask to the thresholded cell
115     digit = cv2.bitwise_and(thresh, thresh, mask=mask)
116
117     # check to see if we should visualize the masking step
118     if debug:
119         cv2.imshow("Digit", digit)
120         cv2.waitKey(0)
121
122     # return the digit to the calling function
123     return digit

```

---

Dividing the pixel area of our `mask` by the area of the cell itself (**Lines 106 and 107**) gives us the `percentFilled` value (i.e., how much our cell is “filled up” with white pixels). Given this percentage, we ensure the contour is not simply “noise” (i.e., a very small contour).

Assuming we don't have a noisy cell, **Line 115** applies the `mask` to the thresholded cell. This mask is optionally shown on screen (**Lines 118–120**) and is finally returned to the caller. Three sample results are shown in Figure 8.7:



**Figure 8.7.** Top-left: a warped cell 8 on the *left* and pre-processed 8 on the *right*. Top-right: a warped cell 1 on the *left* and pre-processed 1 on the *right*. Bottom: a warped cell 9 on the *left* and pre-processed 9 on the *right*.

Great job implementing the digit extraction pipeline!

### 8.2.9 Implementing Our OpenCV Sudoku Puzzle Solver

At this point, we're armed with the following components:

- Our custom **SudokuNet model** trained on the MNIST dataset of digits and **residing on disk** ready for use
- A **means to extract the sudoku puzzle board** and apply a perspective transform
- A **pipeline to extract digits within individual cells** of the sudoku puzzle or ignore ones that we consider to be noise
- The **py-sudoku puzzle solver** (<http://pyimg.co/3ipyb> [27]) **installed** in our Python virtual environment, which saves us from having to engineer an algorithm from hand and lets us focus solely on the computer vision challenge

We are now ready to put each of the pieces together to build a working OpenCV sudoku solver!

Open the `solve_sudoku_puzzle.py` file, and let's complete our sudoku solver project:

```
1 # import the necessary packages
2 from pyimagesearch.sudoku import extract_digit
3 from pyimagesearch.sudoku import find_puzzle
4 from tensorflow.keras.preprocessing.image import img_to_array
5 from tensorflow.keras.models import load_model
```

---

```

6  from sudoku import Sudoku
7  import numpy as np
8  import argparse
9  import imutils
10 import cv2
11
12 # construct the argument parser and parse the arguments
13 ap = argparse.ArgumentParser()
14 ap.add_argument("-m", "--model", required=True,
15     help="path to trained digit classifier")
16 ap.add_argument("-i", "--image", required=True,
17     help="path to input sudoku puzzle image")
18 ap.add_argument("-d", "--debug", type=int, default=-1,
19     help="whether or not we are visualizing each step of the pipeline")
20 args = vars(ap.parse_args())

```

---

As with nearly all Python scripts, we have a selection of imports to get the party started.

These include our custom computer vision helper functions: `extract_digit` and `find_puzzle`. We'll use the Keras/TensorFlow `load_model` method to grab our trained SudokuNet model from disk and load it into memory.

The `sudoku` import is made possible by py-sudoku (<http://pyimg.co/3ipyb> [27]), which we've previously installed.

Let's define three command line arguments (<http://pyimg.co/vsapz> [36]):

- `--model`: The path to our trained digit classifier generated while following the instructions in “*Training Our Sudoku Digit Recognizer with Keras and TensorFlow*” (Section 8.2.6)
- `--image`: Your path to a sudoku puzzle photo residing on disk.
- `--debug`: A flag indicating whether to show intermediate pipeline step debugging visualizations

As we're now equipped with imports and our `args` dictionary, let's load both our (1) digit classifier `model` and (2) input `--image` from disk:

---

```

22 # load the digit classifier from disk
23 print("[INFO] loading digit classifier...")
24 model = load_model(args["model"])
25
26 # load the input image from disk and resize it
27 print("[INFO] processing image...")
28 image = cv2.imread(args["image"])
29 image = imutils.resize(image, width=600)

```

---

From there, we'll find our puzzle and prepare to isolate the cells therein:

---

```

31 # find the puzzle in the image and then
32 (puzzleImage, warped) = find_puzzle(image, debug=args["debug"] > 0)
33
34 # initialize our 9x9 sudoku board
35 board = np.zeros((9, 9), dtype="int")
36
37 # a sudoku puzzle is a 9x9 grid (81 individual cells), so we can
38 # infer the location of each cell by dividing the warped image
39 # into a 9x9 grid
40 stepX = warped.shape[1] // 9
41 stepY = warped.shape[0] // 9
42
43 # initialize a list to store the (x, y)-coordinates of each cell
44 # location
45 cellLocs = []

```

---

Here, we:

- Find the sudoku puzzle in the input --image via our `find_puzzle` helper (**Line 32**)
- Initialize our sudoku board — a  $9 \times 9$  array (**Line 35**)
- Infer the step size for each of the cells by simple division (**Lines 40 and 41**)
- Initialize a list to hold the  $(x, y)$ -coordinates of cell locations (**Line 45**)

And now, let's begin a nested loop over rows and columns of the sudoku board:

---

```

47 # loop over the grid locations
48 for y in range(0, 9):
49     # initialize the current list of cell locations
50     row = []
51
52     for x in range(0, 9):
53         # compute the starting and ending (x, y)-coordinates of the
54         # current cell
55         startX = x * stepX
56         startY = y * stepY
57         endX = (x + 1) * stepX
58         endY = (y + 1) * stepY
59
60         # add the (x, y)-coordinates to our cell locations list
61         row.append((startX, startY, endX, endY))

```

---

Accounting for *every cell* in the sudoku puzzle, we loop over rows (**Line 48**) and columns (**Line 52**) in a nested fashion.

Inside, we use our step values to determine the start/end ( $x, y$ )-coordinates of the *current cell* (**Lines 55–58**).

**Line 61** appends the coordinates as a tuple to this particular `row`. Each row will have nine entries (9 x 4-tuples).

Now we're ready to crop out the `cell` and recognize the digit therein (if one is present):

---

```

63      # crop the cell from the warped transform image and then
64      # extract the digit from the cell
65      cell = warped[startY:endY, startX:endX]
66      digit = extract_digit(cell, debug=args["debug"] > 0)
67
68      # verify that the digit is not empty
69      if digit is not None:
70          # resize the cell to 28x28 pixels and then prepare the
71          # cell for classification
72          roi = cv2.resize(digit, (28, 28))
73          roi = roi.astype("float") / 255.0
74          roi = img_to_array(roi)
75          roi = np.expand_dims(roi, axis=0)
76
77          # classify the digit and update the sudoku board with the
78          # prediction
79          pred = model.predict(roi).argmax(axis=1)[0]
80          board[y, x] = pred
81
82      # add the row to our cell locations
83      cellLocs.append(row)

```

---

Step by step, we proceed to:

- Crop the `cell` from the transformed image and then extract the `digit` (**Lines 65 and 66**)
- If the `digit` is `not None`, then we know there is an actual `digit` in the `cell` (rather than an empty space), at which point we:
  - Pre-process the digit `roi` in the same manner that we did for training (**Lines 72–75**)
  - Classify the digit `roi` with SudokuNet (**Line 79**)
  - Update the sudoku puzzle `board` array with the predicted value of the cell (**Line 80**)
- Add the `row`'s ( $x, y$ )-coordinates to the `cellLocs` list (**Line 83**) — the last line of our nested loop over rows and columns

And now, let's **solve the sudoku puzzle with py-sudoku**:

---

```

85 # construct a sudoku puzzle from the board
86 print("[INFO] OCR'd sudoku board:")
87 puzzle = Sudoku(3, 3, board=board.tolist())
88 puzzle.show()
89
90 # solve the sudoku puzzle
91 print("[INFO] solving sudoku puzzle...")
92 solution = puzzle.solve()
93 solution.show_full()

```

---

As you can see, first, we display the sudoku puzzle board as it was interpreted via OCR (**Lines 87 and 88**).

Then, we make a call to `puzzle.solve` to solve the sudoku puzzle (**Line 92**). And again, this is where the py-sudoku package does the mathematical algorithm to solve our puzzle.

We go ahead and print out the solved puzzle in our terminal (**Line 93**)

And of course, what fun would this project be if we didn't visualize the solution on the puzzle image itself? Let's do that now:

---

```

95 # loop over the cell locations and board
96 for (cellRow, boardRow) in zip(cellLocs, solution.board):
97     # loop over individual cell in the row
98     for (box, digit) in zip(cellRow, boardRow):
99         # unpack the cell coordinates
100        startX, startY, endX, endY = box
101
102        # compute the coordinates of where the digit will be drawn
103        # on the output puzzle image
104        textX = int((endX - startX) * 0.33)
105        textY = int((endY - startY) * -0.2)
106        textX += startX
107        textY += endY
108
109        # draw the result digit on the sudoku puzzle image
110        cv2.putText(puzzleImage, str(digit), (textX, textY),
111                    cv2.FONT_HERSHEY_SIMPLEX, 0.9, (0, 255, 255), 2)
112
113    # show the output image
114    cv2.imshow("Sudoku Result", puzzleImage)
115    cv2.waitKey(0)

```

---

To annotate our image with the solution numbers, we simply:

- Loop over cell locations and the board (**Lines 96–98**)
- Unpack cell coordinates (**Line 100**)

- Compute the coordinates of where text annotation will be drawn (**Lines 104–107**)
- Draw each output digit on our puzzle board photo (**Lines 110 and 111**)
- Display our solved sudoku puzzle image (**Line 114**) until any key is pressed (**Line 115**)

Nice job!

Let's kick our project into gear in the next section. You'll be very impressed with your hard work!

### 8.2.10 OpenCV Sudoku Puzzle Solver OCR Results

We are now ready to put our OpenCV sudoku puzzle solver to the test!

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

---

```
$ python solve_sudoku_puzzle.py --model output/digit_classifier.h5 \
    --image sudoku_puzzle.jpg
[INFO] loading digit classifier...
[INFO] processing image...
[INFO] OCR'd sudoku board:
+-----+-----+-----+
| 8     | 1     | 9     | |
| 5     | 8     | 7     | 1     |
| 4     | 9     | 7     |
+-----+-----+-----+
| 6     | 7     | 1     | 2     | |
| 5     | 8     | 6     | 1     | 7     |
| 1     | 5     | 2     | 9     |
+-----+-----+-----+
| 7     | 4     | 6     | |
| 8     | 3     | 9     | 4     |
| 3     | 5     | 8     |
+-----+-----+-----+
[INFO] solving sudoku puzzle...

-----
9x9 (3x3) SUDOKU PUZZLE
Difficulty: SOLVED
-----
+-----+-----+-----+
| 8 7 2 | 4 1 3 | 5 6 9 |
| 9 5 6 | 8 2 7 | 3 1 4 |
| 1 3 4 | 6 9 5 | 7 8 2 |
+-----+-----+-----+
| 4 6 9 | 7 3 1 | 8 2 5 |
| 5 2 8 | 9 6 4 | 1 3 7 |
| 7 1 3 | 5 8 2 | 4 9 6 |
```

2	9	7	1	4	8	6	5	3
6	8	5	3	7	9	2	4	1
3	4	1	2	5	6	9	7	8

You can see the solved sudoku in Figure 8.8.

**As you can see, we have successfully solved the sudoku puzzle using OpenCV, OCR, and deep learning!**

And now, if you're the betting type, you could challenge a friend or significant other to see who can solve 10 sudoku puzzles the fastest on your next transcontinental airplane ride! Just don't get caught snapping a few photos!



**Figure 8.8.** You'll have to resist the temptation to say "Bingo!" (wrong game) when you achieve this solved sudoku puzzle result using OpenCV, OCR, and Keras/TensorFlow.

### 8.2.11 Credits

This chapter was inspired by Aakash Jhawar and by Part 1 (<http://pyimg.co/tq5yz> [31]) and Part 2 (<http://pyimg.co/hwvq0> [37]) of his sudoku puzzle solver. Additionally, you'll note that I **used the same sample sudoku puzzle board that Aakash did**, not out of laziness, but to demonstrate how the *same* puzzle can be solved with *different* computer vision and image processing techniques.

I enjoyed Aakash Jhawar's articles and recommend readers like you to check them out as well (*especially* if you want to implement a sudoku solver from scratch rather than using the `py-sudoku` library).

## 8.3 Summary

In this chapter, you learned how to implement a sudoku puzzle solver using OpenCV, deep learning, and OCR.

To find and locate the sudoku puzzle board in the image, we utilized OpenCV and basic image processing techniques, including blurring, thresholding, and contour processing, just to name a few. To OCR the digits on the sudoku board, we trained a custom digit recognition model using Keras and TensorFlow. Combining the sudoku board locator with our digit OCR model allowed us to make quick work of solving the actual sudoku puzzle.

## Chapter 9

# Automatically OCR'ing Receipts and Scans

In this chapter, you will learn how to use Tesseract and OpenCV to build an automatic receipt scanner. We'll use OpenCV to build the actual image processing component of the system, including:

- Detecting the receipt in the image
- Finding the four corners of the receipt
- And finally applying a perspective transform to obtain a *top-down*, bird's-eye view of the receipt

From there, we will use Tesseract to OCR the receipt itself and parse out each item, line-by-line, including both the item description and price.

If you're a business owner (like me) and need to report your expenses to your accountant, or if your job requires that you meticulously track your expenses for reimbursement, then you know how frustrating, tedious, and annoying it is to track your receipts. It's hard to believe in this day of age that purchases are *still* tracked via a tiny, fragile piece of paper!

Perhaps in the future, it will become less tedious to track and report our expenses. But until then, receipt scanners can save us a bunch of time and avoid the frustration of manually cataloging purchases.

This chapter's receipt scanner project serves as a starting point for building a full-fledged receipt scanner application. Using this chapter as a starting point — and then extend it by adding a GUI, integrating it with a mobile app, etc.

Let's get started!

## 9.1 Chapter Learning Objectives

In this chapter, you will learn:

- How to use OpenCV to detect, extract, and transform a receipt in an input image
- How to use Tesseract to OCR the receipt, line-by-line
- See a real-world application of how choosing the correct Tesseract PSM can lead to better results

## 9.2 OCR'ing Receipts with OpenCV and Tesseract

In the first part of this chapter, we will review our directory structure for our receipt scanner project.

We'll then review our receipt scanner implementation, line-by-line. Most importantly, I'll show you which Tesseract PSM to use when building a receipt scanner, such that you can easily detect and extract each **item** and **price** from the receipt.

Finally, we'll wrap up this chapter with a discussion of our results.

### 9.2.1 Project Structure

Let's start by reviewing our project directory structure:

---

```
|-- scan_receipt.py
|-- whole_foods.png
```

---

We have only one script to review today, `scan_receipt.py`, which will contain our receipt scanner implementation.

The `whole_foods.png` image is a photo I took of my receipt when I went to Whole Foods, a grocery store chain in the United States. We'll use our `scan_receipt.py` script to detect the receipt in the input image and then extract each item and price from the receipt.

### 9.2.2 Implementing Our Receipt Scanner

Before we dive into the implementation of our receipt scanner, let's first review the basic algorithm that we'll be implementing. When presented with an input image containing a receipt, we will:

- i. Apply edge detection to reveal the outline of the receipt against the background (this assumes that we'll have *sufficient contrast* between the background and foreground, otherwise we won't be able to detect the receipt)
- ii. Detect contours in the edge map
- iii. Loop over all contours and find the largest contour with four vertices (since a receipt is rectangular and will have four corners)
- iv. Apply a perspective transform, yielding a *top-down*, bird's-eye view of the receipt (required to improve OCR accuracy)
- v. Apply the Tesseract OCR engine with `--psm 4` to the top-down transform of the receipt, allowing us to OCR the receipt line-by-line
- vi. Use regular expressions to parse out the item name and price
- vii. Finally, display the results to our terminal

That sounds like a lot of steps, but as you'll see, we can accomplish all of them in less than 120 lines of code (including comments).

With that said, let's dive into the implementation. Open the `scan_receipt.py` file in your project directory structure, and let's get to work:

---

```
1 # import the necessary packages
2 from imutils.perspective import four_point_transform
3 import pytesseract
4 import argparse
5 import imutils
6 import cv2
7 import re
```

---

We start by importing our required Python packages on **Lines 2–7**. These imports most notably include:

- `four_point_transform`: Applies a perspective transform to obtain a *top-down*, bird's-eye view of an input ROI. We used this function in our previous chapter when obtaining a top-down view of a Sudoku board (such that we can automatically solve the puzzles) — we'll be doing the same here today, only with a receipt instead of a Sudoku puzzle.
- `pytesseract`: Provides an interface to the Tesseract OCR engine.
- `cv2`: Our OpenCV bindings

- `re`: Python's regular expression package (<http://pyimg.co/bsmdc> [38]) will allow us to easily parse out the item names and associated prices of each line of the receipt.

Next up, we have our command line arguments:

---

```

9  # construct the argument parser and parse the arguments
10 ap = argparse.ArgumentParser()
11 ap.add_argument("-i", "--image", required=True,
12                 help="path to input receipt image")
13 ap.add_argument("-d", "--debug", type=int, default=-1,
14                 help="whether or not we are visualizing each step of the pipeline")
15 args = vars(ap.parse_args())

```

---

Our script requires one command line argument, followed by an optional one:

- `--image`: The path to our input image contains the receipt we want to OCR (in this case, `whole_foods.png`). You can supply your receipt image here as well.
- `--debug`: An integer value used to indicate whether or not we want to display debugging images through our pipeline, including the output of edge detection, receipt detection, etc.

In particular, if you cannot find a receipt in an input image, it's likely due to either the edge detection process failing to detect the edges of the receipt: meaning that you need to fine-tune your Canny edge detection parameters or use a different method (e.g., thresholding, the Hough line transform, etc.). Another possibility is that the contour approximation step fails to find the four corners of the receipt.

If and when those situations happen, supplying a positive value for the `--debug` command line argument will show you the output of the steps, allowing you to debug the problem, tweak the parameters/algorithms, and then continue.

Next, let's load our `--input` image from disk and examine its spatial dimensions:

---

```

17 # load the input image from disk, resize it, and compute the ratio
18 # of the *new* width to the *old* width
19 orig = cv2.imread(args["image"])
20 image = orig.copy()
21 image = imutils.resize(image, width=500)
22 ratio = orig.shape[1] / float(image.shape[1])

```

---

Here we load our original (`orig`) image from disk and then make a clone. We need to clone the input image such that we have the *original* image to apply the perspective transform to,

but we can apply our actual image processing operations (i.e., edge detection, contour detection, etc.) to the `image`.

We resize our `image` to have a width of 500 pixels (thereby acting as a form of noise reduction) and then compute the `ratio` of the *new* width to the *old* width. This `ratio` value will be used when we need to apply a perspective transform to the `orig` image.

Let's start applying our image processing pipeline to the `image` now:

---

```

24 # convert the image to grayscale, blur it slightly, and then apply
25 # edge detection
26 gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
27 blurred = cv2.GaussianBlur(gray, (5, 5), 0)
28 edged = cv2.Canny(blurred, 75, 200)
29
30 # check to see if we should show the output of our edge detection
31 # procedure
32 if args["debug"] > 0:
33     cv2.imshow("Input", image)
34     cv2.imshow("Edged", edged)
35     cv2.waitKey(0)

```

---

Here we perform edge detection by converting the image to grayscale, blurring it using a  $5 \times 5$  Gaussian kernel (to reduce noise), and then applying edge detection using the Canny edge detector.

If we have our `--debug` command line argument set, we will display both the input image and the output edge map on our screen.

Figure 9.1 shows our input image (*left*), followed by our output edge map (*right*). Notice how our edge map *clearly shows* the outline of the receipt in the input image.

Given our edge map, let's detect contours in the `edged` image and process them:

---

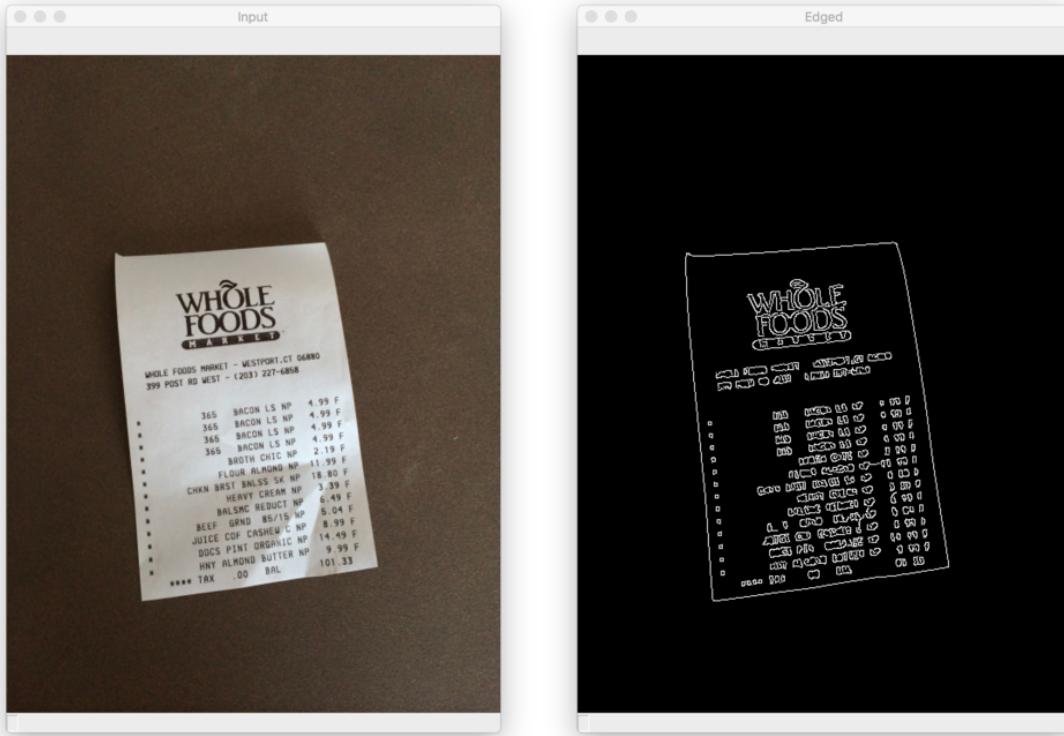
```

37 # find contours in the edge map and sort them by size in descending
38 # order
39 cnts = cv2.findContours(edged.copy(), cv2.RETR_EXTERNAL,
40                         cv2.CHAIN_APPROX_SIMPLE)
41 cnts = imutils.grab_contours(cnts)
42 cnts = sorted(cnts, key=cv2.contourArea, reverse=True)

```

---

Take note of **Line 42** where we are sorting our contours according to their area (size) from largest to smallest. **This sorting step is important as we're making the assumption that the *largest contour* in the input image *with four corners* is our receipt.**



**Figure 9.1.** *Left:* Our input image containing the receipt we are going to OCR. *Right:* Applying edge detection reveals the receipt in the input image.

The sorting step takes care of our first requirement. But how do we know if we've found a contour that has four vertices?

The following code block answers that question:

---

```

44 # initialize a contour that corresponds to the receipt outline
45 receiptCnt = None
46
47 # loop over the contours
48 for c in cnts:
49     # approximate the contour
50     peri = cv2.arcLength(c, True)
51     approx = cv2.approxPolyDP(c, 0.02 * peri, True)
52
53     # if our approximated contour has four points, then we can
54     # assume we have found the outline of the receipt
55     if len(approx) == 4:
56         receiptCnt = approx
57         break
58
59 # if the receipt contour is empty then our script could not find the
60 # outline and we should be notified

```

---

```

61 if receiptCnt is None:
62     raise Exception(("Could not find receipt outline. "
63                     "Try debugging your edge detection and contour steps."))

```

---

**Line 45** initializes a variable to store the contour that corresponds to our receipt. We then start looping over all detected contours on **Line 48**.

**Lines 50 and 51** approximate the contour by reducing the number of points in the contour, thereby simplifying the shape.

**Lines 55–57** check to see if we've found a contour with four points. If so, we can safely assume that we've found our receipt as this is the *largest contour* with *four vertices*. Once we've found the contour, we store it in `receiptCnt` and `break` from the loop.

**Lines 61–63** provide a graceful way for our script exit if our receipt was not found. Typically, this happens when there is a problem during the edge detection phase of our script. Due to insufficient lighting conditions or simply not having enough contrast between the receipt and the background, the edge map may be “broken” by having gaps or holes in it.

When that happens, the contour detection process does not “see” the receipt as a four-corner object. Instead, it sees a strange polygon object, and thus the receipt is not detected.

If and when that happens, be sure to use the `--debug` command line argument to visually inspect your edge map's output.

With our receipt contour found, let's apply our perspective transform to the image:

---

```

65 # check to see if we should draw the contour of the receipt on the
66 # image and then display it to our screen
67 if args["debug"] > 0:
68     output = image.copy()
69     cv2.drawContours(output, [receiptCnt], -1, (0, 255, 0), 2)
70     cv2.imshow("Receipt Outline", output)
71     cv2.waitKey(0)
72
73 # apply a four-point perspective transform to the *original* image to
74 # obtain a top-down bird's-eye view of the receipt
75 receipt = four_point_transform(orig, receiptCnt.reshape(4, 2) * ratio)
76
77 # show transformed image
78 cv2.imshow("Receipt Transform", imutils.resize(receipt, width=500))
79 cv2.waitKey(0)

```

---

If we are in debug mode, **Lines 67–71** draw the outline of the receipt on our `output` image. We then display that output image on our screen to validate that the receipt was detected correctly (Figure 9.2, *left*).

A *top-down*, bird's-eye view of the receipt is accomplished on **Line 75**. Note that we are applying the transform to the higher resolution `orig` image — *why is that?*

First and foremost, the `image` variable has already had edge detection and contour processing applied to it. Using a perspective transform the `image` and then OCR'ing it wouldn't lead to correct results; all we would get is noise.

**Instead, we seek the *high-resolution* version of the receipt.** Hence we apply the perspective transform to the `orig` image. To do so, we need to multiply our `receiptCnt` ( $x, y$ )-coordinates by our `ratio`, thereby scaling the coordinates back to the `orig` spatial dimensions.

To validate that we've computed the *top-down*, bird's-eye view of the original image, we display the high-resolution receipt on our screen on **Lines 78 and 79** (Figure 9.2, right).



**Figure 9.2.** Left: Drawing the receipt's outline/contour, indicating we have successfully found the receipt in the input image. Right: Applying a *top-down* perspective transform of the receipt such that we can OCR it with Tesseract.

Given our *top-down* view of the receipt, we can now OCR it:

```

82 # the text is *concatenated across the row* (additionally, for your
83 # own images you may need to apply additional processing to cleanup
84 # the image, including resizing, thresholding, etc.)
85 options = "--psm 4"
86 text = pytesseract.image_to_string(
87     cv2.cvtColor(receipt, cv2.COLOR_BGR2RGB),
88     config=options)
89
90 # show the raw output of the OCR process
91 print("[INFO] raw output:")
92 print("====")
93 print(text)
94 print("\n")

```

---

**Lines 85–88** use Tesseract to OCR receipt, passing in the `--psm 4` mode. Using `--psm 4` allows us to OCR the receipt line-by-line. Each line will include the item name and the item price.

**Lines 91–94** then show the raw text after applying OCR.

However, the *problem* is that Tesseract has no idea what an *item* is on the receipt versus just the name of the grocery store, address, telephone number, and all the other information you typically find on a receipt.

That raises the question — *how do we parse out the information we do not need, leaving us with just the item names and prices?*

The answer is to leverage regular expressions:

```

96 # define a regular expression that will match line items that include
97 # a price component
98 pricePattern = r'([0-9]+\.[0-9]+)'
99
100 # show the output of filtering out *only* the line items in the
101 # receipt
102 print("[INFO] price line items:")
103 print("====")
104
105 # loop over each of the line items in the OCR'd receipt
106 for row in text.split("\n"):
107     # check to see if the price regular expression matches the current
108     # row
109     if re.search(pricePattern, row) is not None:
110         print(row)

```

---

If you've never used regular expressions before, they are a special tool that allows us to define a pattern of text. The regular expression library (in Python, the library is `re`) then matches all text to this pattern.

**Line 98** defines our `pricePattern`. This pattern will match any number of occurrences of the digits 0–9, followed by the `.` character (meaning the decimal separator in a price value), followed again by any number of the digits 0–9.

For example, this `pricePattern` would match the text `$9.75` but would not match the text `7600` because the text `7600` does not contain a decimal point separator.

If you are new to regular expressions or simply need a refresher on them, I suggest reading the following series by RealPython: <http://pyimg.co/blheh> [39].

**Line 106** splits our raw OCR'd `text` and allows us to loop over each line individually.

For each line, we check to see if the `row` matches our `pricePattern` (**Line 109**). If so, we know we've found a row that contains an item and price, so we print the row to our terminal (**Line 110**).

Congrats on building your first receipt scanner OCR application!

### 9.2.3 Receipt Scanner and OCR Results

Now that we've implemented our `scan_receipt.py` script, let's put it to work. Open a terminal and execute the following command:

---

```
$ python scan_receipt.py --image whole_foods.png
[INFO] raw output:
=====
WHOLE
FOODS

WHOLE FOODS MARKET - WESTPORT, CT 06880
399 POST RD WEST - (203) 227-6858

365 BACON LS NP 4.99
BROTH CHIC NP 4.18

FLOUR ALMOND NP 11.99

CHKN BRST BNLSS SK NP 18.80
HEAVY CREAM NP 3 7

BALSMC REDUCT NP 6.49

BEEF GRND 85/15 NP 5.04
```

```
JUICE COF CASHEW C NP 8.99
DOCS PINT ORGANIC NP 14.49
HNY ALMOND BUTTER NP 9.99
eee TAX .00 BAL 101.33
```

---

The raw output of the Tesseract OCR engine can be seen in our terminal. By specifying `--psm 4`, Tesseract has been able to OCR the receipt line-by-line, capturing both items:

- i. name/description
- ii. price

However, there is a bunch of other “noise” in the output, including the grocery store’s name, address, phone number, etc. **How do we parse out this information, leaving us with only the items and their prices?**

The answer is to use regular expressions which filter on lines that have numerical values similar to prices — the output of these regular expressions can be seen below:

---

```
[INFO] price line items:
=====
365 BACON LS NP 4.99
BROTH CHIC NP 4.18
FLOUR ALMOND NP 11.99
CHKN BRST BNLSS SK NP 18.80
BALSMC REDUCT NP 6.49
BEEF GRND 85/15 NP 5.04
JUICE COF CASHEW C NP 8.99
DOCS PINT ORGANIC NP 14.49
HNY ALMOND BUTTER NP 9.99
eee TAX .00 BAL 101.33
```

---

By using regular expressions we’ve been able to extract only the items and prices, including the final balance due.

Our receipt scanner application is an important implementation to see as it shows how OCR can be combined with a bit of text processing to extract the data of interest. There’s an entire computer science field dedicated to text processing called natural language processing (NLP).

Just like computer vision is the advanced study of how to write software that can understand what’s in an image, NLP seeks to do the same, only for text. Depending on what you’re trying to build with computer vision and OCR, you may want to spend a few weeks to a few months just familiarizing yourself with NLP — that knowledge will better help you understand how to process the text returned from an OCR engine.

### 9.3 Summary

In this chapter, you learned how to implement a basic receipt scanner using OpenCV and Tesseract. Our receipt scanner implementation required basic image processing operations to detect the receipt, including:

- Edge detection
- Contour detection
- Contour filtering using arc length and approximation

From there, we used Tesseract, and most importantly, `--psm 4`, to OCR the receipt. By using `--psm 4`, we were able to extract each item from the receipt, line-by-line, including the item name and the cost of that particular item.

The most significant limitation of our receipt scanner is that it requires:

- i. Sufficient contrast between the receipt and the background
- ii. All four corners of the receipt to be visible in the image

If any of these cases do not hold, our script will not find the receipt.

In our next chapter, we'll look at another popular application of OCR, automatic license/number plate recognition.

## Chapter 10

# OCR'ing Business Cards

In our previous chapter, we learned how to automatically OCR and scan receipts by:

- i. Detecting the receipt in the input image
- ii. Applying a perspective transform to obtain a *top-down* view of the receipt
- iii. Utilizing Tesseract to OCR the text on the receipt
- iv. Applying regular expressions to extract the price data

In this chapter, we're going to use a very similar workflow, but this time apply it to business card OCR. **More specifically, we'll learn how to extract the name, title, phone number, and email address from a business card.**

You'll then be able to extend this implementation to your projects.

### 10.1 Chapter Learning Objectives

In this chapter, you will:

- i. Learn how to detect business cards in images
- ii. Apply OCR to a business card image
- iii. Utilize regular expressions to extract:
  - a. Name
  - b. Job title
  - c. Phone number
  - d. Email address

## 10.2 Business Card OCR

In the first part of this chapter, we will review our project directory structure. We'll then implement a simple yet effective Python script to allow us to OCR a business card.

We'll wrap up this chapter with a discussion of our results, along with the next steps.

### 10.2.1 Project Structure

Before we get too far, let's take a look at our project directory structure:

---

```
|-- larry_page.png
|-- ocr_business_card.py
|-- tony_stark.png
```

---

We only have a single Python script to review, `ocr_business_card.py`. This script will load example business card images (i.e., `larry_page.png` and `tony_stark.png`), OCR them, and then output the name, job title, phone number, and email address from the business card.

Best of all, we'll be able to accomplish our goal in under 120 lines of code (including comments)!

### 10.2.2 Implementing Business Card OCR

We are now ready to implement our business card OCR script! Open the `ocr_business_card.py` file in our project directory structure and insert the following code:

---

```
1 # import the necessary packages
2 from imutils.perspective import four_point_transform
3 import pytesseract
4 import argparse
5 import imutils
6 import cv2
7 import re
```

---

Our imports here are similar to the ones in our previous chapter on OCR'ing receipts.

We need our `four_point_transform` function to obtain a *top-down*, bird's-eye view of the business card. Obtaining this view typically yields higher OCR accuracy.

The `pytesseract` package is used to interface with the Tesseract OCR engine. We then have Python's regular expression library, `re`, which will allow us to parse the names, job titles, email addresses, and phone numbers from business cards.

With the imports taken care of, we can move on to command line arguments:

---

```

9 # construct the argument parser and parse the arguments
10 ap = argparse.ArgumentParser()
11 ap.add_argument("-i", "--image", required=True,
12                 help="path to input image")
13 ap.add_argument("-d", "--debug", type=int, default=-1,
14                 help="whether or not we are visualizing each step of the pipeline")
15 ap.add_argument("-c", "--min-conf", type=int, default=0,
16                 help="minimum confidence value to filter weak text detection")
17 args = vars(ap.parse_args())

```

---

Our first command line argument, `--image`, is the path to our input image on disk. We assume that this image contains a business card with sufficient contrast between the foreground and background, ensuring we can apply edge detection and contour processing to successfully extract the business card.

We then have two optional command line arguments, `--debug` and `--min-conf`. The `--debug` command line argument is used to indicate if we are debugging our image processing pipeline and showing more of the processed images on our screen (useful for when you can't determine why a business card was detected or not).

We then have `--min-conf`, which is the minimum confidence (on a scale of 0–100) required for successful text detection. You can increase `--min-conf` to prune out weak text detections.

Let's now load our input image from disk:

---

```

19 # load the input image from disk, resize it, and compute the ratio
20 # of the *new* width to the *old* width
21 orig = cv2.imread(args["image"])
22 image = orig.copy()
23 image = imutils.resize(image, width=600)
24 ratio = orig.shape[1] / float(image.shape[1])

```

---

Here we load our input `--image` from disk and then clone it. We make it a clone so that we can extract the original high-resolution version of the business card after contour processing.

We then resize our `image` to have a width of 600px and then compute the ratio of the *new* width to the *old* width (a requirement for when we want to obtain a *top-down* view of the original high-resolution business card).

We continue our image processing pipeline below.

---

```

26 # convert the image to grayscale, blur it, and apply edge detection
27 # to reveal the outline of the business card

```

---

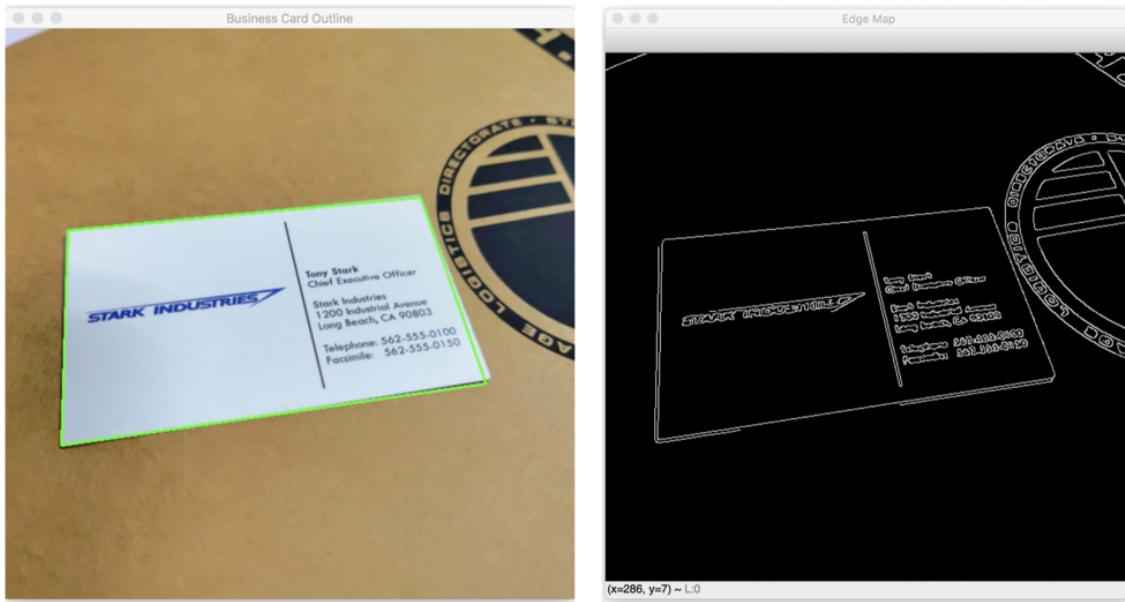
```

28 gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
29 blurred = cv2.GaussianBlur(gray, (5, 5), 0)
30 edged = cv2.Canny(blurred, 30, 150)
31
32 # detect contours in the edge map, sort them by size (in descending
33 # order), and grab the largest contours
34 cnts = cv2.findContours(edged.copy(), cv2.RETR_EXTERNAL,
35     cv2.CHAIN_APPROX_SIMPLE)
36 cnts = imutils.grab_contours(cnts)
37 cnts = sorted(cnts, key=cv2.contourArea, reverse=True) [:5]
38
39 # initialize a contour that corresponds to the business card outline
40 cardCnt = None

```

---

First, we take our original `image` and then convert it to grayscale, blur it, and then apply edge detection, the result of which can be seen in Figure 10.1.



**Figure 10.1.** *Left:* Our input image containing a business card. *Right:* Computing the edge map of the input image reveals the outlines of the business card.

Note that the outline/border of the business card is visible on the edge map. Suppose there are any gaps in the edge map. In that case, the business card will *not* be detectable via our contour processing technique, so you may need to tweak the parameters to the Canny edge detector or capture your image in an environment with better lighting conditions.

From there, we detect contours and sort them in descending order (largest to smallest) based on the area of the computed contour. Our assumption here will be the business card contour will be one of the largest detected contours, hence this operation.

We also initialize `cardCnt` (**Line 40**), which is the contour that corresponds to the business card.

Let's now loop over the largest contours:

---

```

42 # loop over the contours
43 for c in cnts:
44     # approximate the contour
45     peri = cv2.arcLength(c, True)
46     approx = cv2.approxPolyDP(c, 0.02 * peri, True)
47
48     # if this is the first contour we've encountered that has four
49     # vertices, then we can assume we've found the business card
50     if len(approx) == 4:
51         cardCnt = approx
52         break
53
54     # if the business card contour is empty then our script could not
55     # find the outline of the card, so raise an error
56     if cardCnt is None:
57         raise Exception(("Could not find receipt outline. "
58                         "Try debugging your edge detection and contour steps."))

```

---

**Lines 45 and 46** perform contour approximation.

If our approximated contour has four vertices, then we can assume that we found the business card. If that happens, we break from the loop and update our `cardCnt`.

If we reach the end of the `for` loop and still haven't found a valid `cardCnt`, then we gracefully exit the script. Remember, we cannot process the business card if one cannot be found in the image!

Our next code block handles showing some debugging images as well as obtaining our *top-down* view of the business card:

---

```

60 # check to see if we should draw the contour of the business card
61 # on the image and then display it to our screen
62 if args["debug"] > 0:
63     output = image.copy()
64     cv2.drawContours(output, [cardCnt], -1, (0, 255, 0), 2)
65     cv2.imshow("Business Card Outline", output)
66     cv2.waitKey(0)
67
68 # apply a four-point perspective transform to the *original* image to
69 # obtain a top-down bird's-eye view of the business card
70 card = four_point_transform(orig, cardCnt.reshape(4, 2) * ratio)
71
72 # show transformed image

```

---

---

```
73 cv2.imshow("Business Card Transform", card)
74 cv2.waitKey(0)
```

---

**Lines 62–66** make a check to see if we are in `--debug` mode, and if so, we draw the contour of the business card on the `output` image.

We then apply a four-point perspective transform to the *original, high-resolution image*, thus obtaining the *top-down*, bird's-eye view of the business card (**Line 70**).

We multiply the `cardCnt` by our computed `ratio` here since `cardCnt` was computed for the reduced image dimensions. Multiplying by `ratio` scales the `cardCnt` back into the dimensions of the `orig` image.

We then display the transformed image to our screen (**Lines 73 and 74**).

With our *top-down* view of the business card obtain, we can move on to OCR'ing it:

---

```
76 # convert the business card from BGR to RGB channel ordering and then
77 # OCR it
78 rgb = cv2.cvtColor(card, cv2.COLOR_BGR2RGB)
79 text = pytesseract.image_to_string(rgb)
80
81 # use regular expressions to parse out phone numbers and email
82 # addresses from the business card
83 phoneNums = re.findall(r'[\+\(\)]?[1-9][0-9]\.\-\(\)]{8,}[0-9]', text)
84 emails = re.findall(r"[a-z0-9]\.\-+_\]+@[a-z0-9]\.\-+_\]+\.[a-z]+", text)
85
86 # attempt to use regular expressions to parse out names/titles (not
87 # necessarily reliable)
88 nameExp = r"^\w'\-,.][^0-9_!;?÷?¿/\\"+=@#$%^&*(){}|~<>;:[\]]{2,}"
89 names = re.findall(nameExp, text)
```

---

**Lines 78 and 79** OCR the business card, resulting in the `text` output.

But the question remains, how are we going to extract the information from the business card itself? **The answer is to utilize regular expressions.**

**Lines 83 and 84** utilize regular expressions to extract phone numbers and email addresses (<http://pyimg.co/eb5kf> [40]) from the `text`, while **Lines 88 and 89** do the same for names and job titles (<http://pyimg.co/oox3v> [41]).

A review of regular expressions is outside the scope of this chapter, but the gist is that they can be used to match particular patterns in text.

For example, a phone number consists of a specific digits pattern, and sometimes includes dashes and parentheses. Email addresses also follow a pattern, including a string of text, followed by an "@" symbol, and then the domain name.

Any time you can reliably guarantee a pattern of text, regular expressions can work quite well. That said, they aren't perfect either, so you may want to look into more advanced natural language processing (NLP) algorithms if you find your business card OCR accuracy is suffering significantly.

The final step here is to display our output to the terminal:

---

```
91 # show the phone numbers header
92 print("PHONE NUMBERS")
93 print("=====")
94
95 # loop over the detected phone numbers and print them to our terminal
96 for num in phoneNums:
97     print(num.strip())
98
99 # show the email addresses header
100 print("\n")
101 print("EMAILS")
102 print("=====")
103
104 # loop over the detected email addresses and print them to our
105 # terminal
106 for email in emails:
107     print(email.strip())
108
109 # show the name/job title header
110 print("\n")
111 print("NAME/JOB TITLE")
112 print("=====")
113
114 # loop over the detected name/job titles and print them to our
115 # terminal
116 for name in names:
117     print(name.strip())
```

---

This final code block loops over the extracted phone numbers (**Lines 96 and 97**), email addresses (**Lines 106 and 107**), and names/job titles (**Lines 116 and 117**), displaying each to our terminal.

Of course, you could take this extracted information, write to disk, save it to a database, etc. Still, for the sake of simplicity (and not knowing your project specifications of business card OCR), we'll leave it as an exercise to you to save the data as you see fit.

### 10.2.3 Business Card OCR Results

We are now ready to apply OCR to business cards. Open a terminal and execute the following command:

---

```
$ python ocr_business_card.py --image tony_stark.png --debug 1
PHONE NUMBERS
=====
562-555-0100
562-555-0150

EMAILS
=====

NAME/JOB TITLE
=====
Tony Stark
Chief Executive Officer

Stark Industries
```

---

Figure 10.2 (*top*) shows the results of our business card localization. Notice how we have correctly detected the business card in the input image.

From there, Figure 10.2 (*bottom*) displays the results of applying a perspective transform of the business card, thus resulting in the *top-down*, bird's-eye view of the image.



**Figure 10.2.** *Top:* An input image containing Tony Stark's business card. *Bottom:* Obtaining a *top-down*, bird's-eye view of the input business card.

Once we have the *top-down* view of the image (typically required to obtain higher OCR accuracy), we can apply Tesseract to OCR it, the results of which can be seen in our terminal output above.

Note that our script has successfully extracted *both* phone numbers on Tony Stark's business card.

No email addresses are reported as there is no email address on the business card.

We then have the name and job title displayed as well. It's interesting to see that we are able to OCR all the text successfully because the text of the name is more distorted than the phone number text. Our perspective transform was able to deal with all the text effectively even though the amount of distortion changes as you go further away from the camera. That's the point of perspective transform and why it's important to the accuracy of our OCR.

Let's try another example image, this one of an old Larry Page (co-founder of Google) business card:

---

```
$ python ocr_business_card.py --image larry_page.png --debug 1
PHONE NUMBERS
=====
650 330-0100
650 618-1499

EMAILS
=====
larry@google.com

NAME / JOB TITLE
=====
Larry Page
CEO

Google
```

---

Figure 10.3 (*top*) displays the output of localizing Page's business card. The *bottom* then shows the *top-down* transform of the image.

This *top-down* transform is passed through Tesseract OCR, yielding the OCR'd text as output. We take this OCR'd text, apply a regular expression, and thus obtain the results above.

Examining the results, you can see that we have successfully extracted Larry Page's two phone numbers, email address, and name/job title from the business card.



**Figure 10.3.** Top: An input image containing Larry Page's business card. Bottom: Obtaining a top-down view of the business card, which we can then OCR.

### 10.3 Summary

In this chapter, you learned how to build a basic business card OCR system. This system was, essentially, an extension of our receipt scanner, but with different regular expressions and text localization strategies.

If you ever need to build a business card OCR system, I recommend that you use this chapter as a starting point, but keep in mind that you may want to utilize more advanced text post-processing techniques, such as true natural language processing (NLP) algorithms, rather than regular expressions.

Regular expressions can work *very well* for email addresses and phone numbers, but for names and job titles that may fail to obtain high accuracy. If and when that time comes, you should consider leveraging NLP as much as possible to improve your results.

## Chapter 11

# OCR'ing License Plates with ANPR/ALPR

In this chapter, you will build a basic automatic license/number plate recognition (ALPR/ANPR) system using OpenCV and Python.

The ALPR/ANPR systems come in all shapes and sizes:

- ALPR/ANPR performed in controlled lighting conditions with predictable license plate types can use basic image processing techniques.
- More advanced ANPR systems utilize dedicated object detectors (e.g., Histogram of Oriented Gradient (HOG) + Linear Support Vector Machine (SVM), Faster R-CNN, Single-Shot Detectors (SSDs), and you only look once (YOLO)) to localize license plates in images.
- State-of-the-art ANPR software utilizes Recurrent Neural Networks (RNNs) and Long Short-Term Memory (LSTM) networks to better OCR text from the license plates themselves.
- And even *more advanced* ANPR systems use specialized neural network architectures to pre-process, and clean images before they are OCR'd, thereby improving ANPR accuracy.

The ALPR/ANPR system is further complicated because it may need to operate in *real-time*.

For example, suppose an ANPR system is mounted on a toll road. It needs to detect the license plate of each car passing by, optical character recognition (OCR) the characters on the plate, and *then* store this information in a database so the owner of the vehicle can be billed for the toll.

Several compounding factors make ANPR incredibly challenging, including *finding a dataset*

*you can use to train a custom ANPR model!* Large, robust ANPR datasets that are used to train state-of-the-art models are closely guarded and rarely (if ever) released publicly:

- These datasets contain sensitive identifying information related to the vehicle, driver, and location.
- ANPR datasets are tedious to curate, requiring an incredible investment of time and staff hours to annotate.
- ANPR contracts with local and federal governments tend to be *highly competitive*. Because of that, it's often not the *trained model* that is valuable, but instead the *dataset* that a given company has curated.

For that reason, you'll see ANPR companies acquired not for their ANPR system *but for the data itself!*

In the remainder of this chapter, we'll be building a basic ALPR/ANPR system. By the end of this chapter, you'll have a template/starting point to use when building your ANPR projects.

## 11.1 Chapter Learning Objectives

In this chapter, you will:

- Discover what ALPR/ANPR is
- Implement a function used to detect license plates in images
- Create a second function to OCR the license plate
- Glue the pieces together and form a basic yet complete ANPR project

## 11.2 ALPR/ANPR and OCR

My first run-in with ALPR/ANPR was about six years ago.

After a grueling three-day marathon consulting project in Maryland, where it did nothing but rain the entire time, I hopped on I-95 to drive back to Connecticut to visit friends for the weekend.

It was a beautiful summer day. Sun shining. Not a cloud in the sky. A soft breeze blowing. Perfect. Of course, I had my windows down, my music turned up, and I had totally zoned out — not a care in the world.

I didn't even notice when I drove past a small gray box discreetly positioned along the side of the highway.

Two weeks later . . . I got the speeding ticket in the mail.

Sure enough, I had unknowingly driven past a speed-trap camera doing 78 miles per hour (mph) in a 65-mph zone.

That speeding camera caught me with my foot on the pedal, quite literally, and it had pictures to prove it too. There it was, clear as day! You could see the license plate number on my old Honda Civic (before it got burnt to a crisp in an electrical fire).

Now, here's the ironic part. I knew exactly how their ALPR/ANPR system worked. I knew which image processing techniques the developers used to automatically localize my license plate in the image and extract the plate number via OCR.

**In this chapter, my goal is to teach you one of the quickest ways to build such an ALPR/ANPR system.**

Using a bit of OpenCV, Python, and Tesseract OCR knowledge, you could help your homeowners' association monitor cars that come and go from your neighborhood.

Or maybe you want to build a camera-based (radar-less) system that determines the speed of cars that drive by your house using a Raspberry Pi (<http://pyimg.co/5xigt> [42]). If the car exceeds the speed limit, you can analyze the license plate, apply OCR to it, and log the license plate number to a database. Such a system could help reduce speeding violations and create better neighborhood safety.

In the first part of this chapter, you'll learn and define what ALPR/ANPR is. From there, we'll review our project structure. I'll then show you how to implement a basic Python class (aptly named `PyImageSearchANPR`) that will **localize license plates in images** and then **OCR the characters**.

We'll wrap up the chapter by examining the results of our ANPR system.

### 11.2.1 What Is Automatic License Plate Recognition?

ALPR/ANPR is a process involving the following steps:

- **Step #1:** Detect and localize a license plate in an input image/frame
- **Step #2:** Extract the characters from the license plate
- **Step #3:** Apply some form of OCR to recognize the extracted characters

**ANPR tends to be an extremely challenging subfield of computer vision due to the vast diversity and assortment of license plate types across states and countries.**

License plate recognition systems are further complicated by:

- Dynamic lighting conditions including reflections, shadows, and blurring
- Fast-moving vehicles
- Obstructions

Additionally, large and robust ANPR datasets for training/testing are difficult to obtain due to:

- i. These datasets containing sensitive, personal information, including the time and location of a vehicle and its driver
- ii. ANPR companies and government entities closely guarding these datasets as proprietary information

Therefore, the first part of an ANPR project is to collect data and amass enough sample plates under various conditions.

Let's assume we don't have a license plate dataset (quality datasets are hard to come by). That rules out deep learning object detection, which means we will have to exercise our traditional computer vision knowledge.

I agree that it would be nice to have a trained object detection model, but today I want *you* to rise to the occasion.

Before long, we'll be able to ditch the training wheels and consider working for a toll technology company, red-light camera integrator, speed ticketing system, or parking garage ticketing firm in which we need 99.97% accuracy.

Given these limitations, we'll be building a *basic* ANPR system that you can use as a *starting point* for your projects.

### 11.2.2 Project Structure

Before we write any code, let's first review our project directory structure:

---

```
|-- license_plates
|   |-- group1
|   |   |-- 001.jpg
|   |   |-- 002.jpg
|   |   |-- 003.jpg
```

---

```

|   |   |-- 004.jpg
|   |   |-- 005.jpg
|   |-- group2
|   |   |-- 001.jpg
|   |   |-- 002.jpg
|   |   |-- 003.jpg
|-- pyimagesearch
|   |-- __init__.py
|   |-- helpers.py
|   |-- anpr
|   |   |-- __init__.py
|   |   |-- anpr.py
|-- ocr_license_plate.py

```

---

The project folder contains:

- `license_plates`: Directory containing two subdirectories of JPEG images
- `anpr.py`: Contains the `PyImageSearchANPR` class responsible for localizing license/number plates and performing OCR
- `ocr_license_plate.py`: Our main driver Python script, which uses our `PyImageSearchANPR` class to OCR entire groups of images

Now that we have the lay of the land, let's walk through our two Python scripts, which locate and OCR groups of license/number plates and display the results.

### 11.2.3 Implementing ALPR/ANPR with OpenCV and OCR

We're ready to start implementing our ALPR/ANPR script.

As I mentioned before, we'll keep our code neat and organized using a Python class appropriately named `PyImageSearchANPR`. This class provides a reusable means for license plate localization and character OCR operations.

Open `anpr.py` and let's get to work reviewing the script:

---

```

1 # import the necessary packages
2 from skimage.segmentation import clear_border
3 import pytesseract
4 import numpy as np
5 import imutils
6 import cv2
7
8 class PyImageSearchANPR:
9     def __init__(self, minAR=4, maxAR=5, debug=False):
10         # store the minimum and maximum rectangular aspect ratio

```

```
11      # values along with whether or not we are in debug mode
12      self.minAR = minAR
13      self.maxAR = maxAR
14      self.debug = debug
```

---

At this point, these imports should feel pretty standard and straightforward. The exception is perhaps scikit-image's `clear_border` function — this method helps clean up the borders of images, thereby improving Tesseract's OCR accuracy.

Our `PyImageSearchANPR` class begins on **Line 8**. The constructor accepts three parameters:

- `minAR`: The minimum aspect ratio used to detect and filter rectangular license plates, which has a default value of 4
- `maxAR`: The maximum aspect ratio of the license plate rectangle, which has a default value of 5
- `debug`: A flag to indicate whether we should display intermediate results in our image processing pipeline

The aspect ratio range (`minAR` to `maxAR`) corresponds to a license plate's typical rectangular dimensions. Keep the following considerations in mind if you need to alter the aspect ratio parameters:

- European and international plates are often longer and not as tall as United States license plates. We're not considering U.S. license plates.
- Sometimes, motorcycles and large dumpster trucks mount their plates sideways; this is a true edge case that would have to be considered for a highly accurate license plate system.
- Some countries and regions allow for multi-line plates with a near 1:1 aspect ratio; again, we won't consider this edge case.

Each of our constructor parameters becomes a class variable on **Lines 12–14** so the methods in the class can access them.

#### 11.2.4 Debugging Our Computer Vision Pipeline

With our constructor ready to go, let's define a helper function to display results at various points in the imaging pipeline when in `debug` mode:

---

```

16 def debug_imshow(self, title, image, waitKey=False):
17     # check to see if we are in debug mode, and if so, show the
18     # image with the supplied title
19     if self.debug:
20         cv2.imshow(title, image)
21
22     # check to see if we should wait for a keypress
23     if waitKey:
24         cv2.waitKey(0)

```

---

Our helper function `debug_imshow` (**Line 16**) accepts three parameters:

- i. `title`: The desired OpenCV window title. Window titles should be unique; otherwise OpenCV will replace the image in the same-titled window rather than creating a new one.
- ii. `image`: The image to display inside the OpenCV graphical user interface (GUI) window.
- iii. `waitKey`: A flag to see if the display should wait for a keypress before completing.

**Lines 19–24** display the debugging `image` in an OpenCV window. Typically, the `waitKey` Boolean will be `False`. However, we have set it to `True` to inspect debugging images and dismiss them when we are ready.

### 11.2.5 Locating Potential License Plate Candidates

Our first ANPR method helps us to find the license plate candidate contours in an image:

---

```

26 def locate_license_plate_candidates(self, gray, keep=5):
27     # perform a blackhat morphological operation which will allow
28     # us to reveal dark regions (i.e., text) on light backgrounds
29     # (i.e., the license plate itself)
30     rectKern = cv2.getStructuringElement(cv2.MORPH_RECT, (13, 5))
31     blackhat = cv2.morphologyEx(gray, cv2.MORPH_BLACKHAT, rectKern)
32     self.debug_imshow("Blackhat", blackhat)

```

---

Our `locate_license_plate_candidates` expects two parameters:

- i. `gray`: This function assumes that the driver script will provide a grayscale image containing a potential license plate.
- ii. `keep`: We'll only return *up to this many* sorted license plate candidate contours.

We're now going to make a **generalization** to help us **simplify our ANPR pipeline**. Let's *assume* from here forward that most license plates have a light background (typically it is highly reflective) and a dark foreground (characters).

I realize there are plenty of cases where this generalization does not hold, but let's continue working on our proof of concept, and we can make accommodations for inverse plates in the future.

**Lines 30 and 31** perform a **blackhat morphological operation** to reveal dark characters (letters, digits, and symbols) against light backgrounds (the license plate itself). As you can see, our kernel has a rectangular shape of 13 pixels wide x 5 pixels tall, which corresponds to the shape of a typical international license plate.

If your `debug` option is on, you'll see a blackhat visualization similar to the one in Figure 11.1 (*bottom-left*). As you can see from above, the license plate characters are visible!



**Figure 11.1.** *Top-left:* Our sample input image containing a license plate that we wish to detect and OCR. *Bottom-left:* OpenCV's blackhat morphological operator highlights the license plate numbers against the rest of the photo of the rear end of the car. You can see that the license plate numbers “pop” as white text against the black background and most of the background noise is washed out. *Top-right:* Perform a closing and threshold operation — notice how the regions where the license plate is located are almost one large white surface. *Bottom-right:* Applying Scharr’s algorithm in the x-direction emphasizes the edges in our blackhat image as another ANPR image processing pipeline step.

In our next step, we'll find regions in the image that are light and *may contain* license plate characters:

---

```

34     # next, find regions in the image that are light
35     squareKern = cv2.getStructuringElement(cv2.MORPH_RECT, (3, 3))

```

---

```

36     light = cv2.morphologyEx(gray, cv2.MORPH_CLOSE, squareKern)
37     light = cv2.threshold(light, 0, 255,
38         cv2.THRESH_BINARY | cv2.THRESH_OTSU) [1]
39     self.debug_imshow("Light Regions", light)

```

---

Using a small square kernel (**Line 35**), we apply a closing operation (**Lines 36**) to fill small holes and identify larger image structures. **Lines 37 and 38** perform a binary threshold on our image using Otsu's method to reveal the light regions in the image that may contain license plate characters.

Figure 11.1 (*top-right*) shows the effect of the closing operation combined with Otsu's inverse binary thresholding. Notice how the regions where the license plate is located are almost one large white surface.

The Scharr gradient will detect edges in the image and emphasize the boundaries of the characters in the license plate:

---

```

41     # compute the Scharr gradient representation of the blackhat
42     # image in the x-direction and then scale the result back to
43     # the range [0, 255]
44     gradX = cv2.Sobel(blackhat, ddepth=cv2.CV_32F,
45         dx=1, dy=0, ksize=-1)
46     gradX = np.absolute(gradX)
47     (minVal, maxVal) = (np.min(gradX), np.max(gradX))
48     gradX = 255 * ((gradX - minVal) / (maxVal - minVal))
49     gradX = gradX.astype("uint8")
50     self.debug_imshow("Schar", gradX)

```

---

Using `cv2.Sobel`, we compute the Scharr gradient magnitude representation in the *x*-direction of our `blackhat` image (**Lines 44 and 45**). We then scale the resulting intensities back to the range  $[0, 255]$  (**Lines 46–49**).

Figure 11.1 (*bottom-right*) demonstrates an emphasis on the edges of the license plate characters. As you can see above, the license plate characters appear noticeably different from the rest of the image.

We can now smooth to group the regions that may contain boundaries to license plate characters:

---

```

52     # blur the gradient representation, applying a closing
53     # operation, and threshold the image using Otsu's method
54     gradX = cv2.GaussianBlur(gradX, (5, 5), 0)
55     gradX = cv2.morphologyEx(gradX, cv2.MORPH_CLOSE, rectKern)
56     thresh = cv2.threshold(gradX, 0, 255,
57         cv2.THRESH_BINARY | cv2.THRESH_OTSU) [1]
58     self.debug_imshow("Grad Thresh", thresh)

```

---

Here we apply a Gaussian blur to the gradient magnitude image (`gradX`) (**Line 54**). Again, we apply a closing operation (**Line 55**) and another binary threshold using Otsu's method (**Lines 56 and 57**).

Figure 11.2 (*top-left*) shows a contiguous white region where the license plate characters are located. At first glance, these results look cluttered. The license plate region is somewhat defined, but there are many other large white regions as well.



**Figure 11.2.** *Top-left:* Blurring, closing, and thresholding operations result in a contiguous white region on top of the license plate/number plate characters. *Top-right:* Erosions and dilations with clean up our thresholded image, making it easier to find our license plate characters for our ANPR system. *Bottom:* After a series of image processing pipeline steps, we can see the region with the license plate characters is one of the larger contours.

Let's see if we can eliminate some of the noise:

---

```

60     # perform a series of erosions and dilations to clean up the
61     # thresholded image
62     thresh = cv2.erode(thresh, None, iterations=2)
63     thresh = cv2.dilate(thresh, None, iterations=2)
64     self.debug_imshow("Grad Erode/Dilate", thresh)

```

---

**Lines 62 and 63** perform a series of erosions and dilations in an attempt to denoise the thresholded image. As you can see in Figure 11.2 (*top-right*), the erosion and dilation

operations cleaned up a lot of noise in the previous result from Figure 11.1. We aren't done yet, though.

Let's add another step to the pipeline, in which we'll put our light region's image to use:

---

```

66      # take the bitwise AND between the threshold result and the
67      # light regions of the image
68      thresh = cv2.bitwise_and(thresh, thresh, mask=light)
69      thresh = cv2.dilate(thresh, None, iterations=2)
70      thresh = cv2.erode(thresh, None, iterations=1)
71      self.debug_imshow("Final", thresh, waitKey=True)

```

---

Back on **Lines 35–38**, we devised a method to highlight lighter regions in the image (keeping in mind our established generalization that license plates will have a light background and dark foreground).

This light image serves as our `mask` for a bitwise-AND between the thresholded result and the image's light regions to reveal the license plate candidates (**Line 68**). We follow with a couple of dilations and an erosion to fill holes and clean up the image (**Lines 69 and 70**).

Our final debugging image is shown in Figure 11.2 (bottom). Notice that the last call to `debug_imshow` overrides `waitKey` to `True`, ensuring that as a user, we can inspect all debugging images up until this point and press a key when we are ready.

You should notice that our license plate contour is not the largest, but it's far from being the smallest. At a glance, I'd say it is the second or third largest contour in the image, and I also notice the plate contour is not touching the edge of the image.

Speaking of contours, let's find and sort them:

---

```

73      # find contours in the thresholded image and sort them by
74      # their size in descending order, keeping only the largest
75      # ones
76      cnts = cv2.findContours(thresh.copy(), cv2.RETR_EXTERNAL,
77                             cv2.CHAIN_APPROX_SIMPLE)
78      cnts = imutils.grab_contours(cnts)
79      cnts = sorted(cnts, key=cv2.contourArea, reverse=True) [:keep]
80
81      # return the list of contours
82      return cnts

```

---

To close out our `locate_license_plate_candidates` method, we:

- Find all contours (**Lines 76–78**)
- Reverse-sort them according to their pixel area while only retaining at most `keep` contours

- Return the resulting sorted and pruned list of `cnts` (**Line 82**).

Take a step back to think about what we've accomplished in this method. We've accepted a grayscale image and used traditional image processing techniques with an emphasis on morphological operations to find a selection of candidate contours that *might* contain a license plate.

I know what you are thinking:

*"Why haven't we applied deep learning object detection to find the license plate? Wouldn't that be easier?"*

While that is perfectly acceptable (and don't get me wrong, I love deep learning!), it is a lot of work to train such an object detector on your own. What we are considering by training our own object detector requires countless hours to annotate thousands of images in your dataset.

**But remember, we didn't have the luxury of a dataset in the first place**, so the method we've developed so far relies on so-called "traditional" image processing techniques.

### 11.2.6 Pruning License Plate Candidates

In this next method, our goal is to find the most likely contour containing a license plate from our set of candidates. Let's see how it works:

---

```

84 def locate_license_plate(self, gray, candidates,
85     clearBorder=False):
86     # initialize the license plate contour and ROI
87     lpCnt = None
88     roi = None
89
90     # loop over the license plate candidate contours
91     for c in candidates:
92         # compute the bounding box of the contour and then use
93         # the bounding box to derive the aspect ratio
94         (x, y, w, h) = cv2.boundingRect(c)
95         ar = w / float(h)

```

---

Our `locate_license_plate` function accepts three parameters:

- i. `gray`: Our input grayscale image
- ii. `candidates`: The license plate contour candidates returned by the previous method in this class

- iii. `clearBorder`: A Boolean indicating whether our pipeline should eliminate any contours that touch the edge of the image

Before we begin looping over the license plate contour candidates, first we initialize variables that will soon hold our license plate contour (`lpCnt`) and license plate region of interest (`roi`) on **Lines 87 and 88**.

Starting on **Line 91**, our loop begins. This loop aims to isolate the contour that contains the license plate and extract the region of interest of the license plate itself. We proceed by determining the bounding box rectangle of the contour, `c` (**Line 94**).

Computing the aspect ratio of the contour's bounding box (**Line 95**) will ensure our contour is the proper rectangular shape of a license plate.

As you can see in the equation, the aspect ratio is a relationship between the rectangle's width and height.

Let's inspect the aspect ratio now and filter out bounding boxes that are not good candidates for license plates:

---

```

97             # check to see if the aspect ratio is rectangular
98             if ar >= self.minAR and ar <= self.maxAR:
99                 # store the license plate contour and extract the
100                # license plate from the gray scale image and then
101                # threshold it
102                lpCnt = c
103                licensePlate = gray[y:y + h, x:x + w]
104                roi = cv2.threshold(licensePlate, 0, 255,
105                                cv2.THRESH_BINARY_INV | cv2.THRESH_OTSU) [1]
```

---

If the contour's bounding box `ar` does not meet our license plate expectations, then there's no more work to do. The `roi` and `lpCnt` will remain as `None`, and it is up to the driver script to handle this scenario.

Hopefully, the aspect ratio is acceptable and falls within the bounds of a typical license plate's `minAR` and `maxAR`. In this case, we assume that we have our winning license plate contour! Let's go ahead and populate `lpCnt` and our `roi`:

- `lpCnt` is set from the current contour, `c` (**Line 102**).
- `roi` is extracted via NumPy slicing (**Line 103**) and subsequently binary inverse thresholded using Otsu's method (**Lines 104 and 105**).

Let's wrap up the `locate_license_plate` method so we can move onto the next phase:

---

```

107      # check to see if we should clear any foreground
108      # pixels that are touching the border of the image
109      # (which typically, not but always, indicates noise)
110      if clearBorder:
111          roi = clear_border(roi)
112
113      # display any debugging information and then break
114      # from the loop early since we have found the license
115      # plate region
116      self.debug_imshow("License Plate", licensePlate)
117      self.debug_imshow("ROI", roi, waitKey=True)
118      break
119
120      # return a 2-tuple of the license plate ROI and the contour
121      # associated with it
122      return (roi, lpCnt)

```

---

If our `clearBorder` flag is set, we can clear any foreground pixels that are touching the border of our license plate ROI (**Lines 110 and 111**). Clearing the foreground pixels touching the license plate ROI helps to eliminate noise that could impact our Tesseract OCR results.

**Lines 116 and 117** display our:

- `licensePlate`: The ROI pre-thresholding and border cleanup (Figure 11.3, *top*)
- `roi`: Our final license plate ROI (Figure 11.3, *bottom*)

Again, notice that the last call to `debug_imshow` of this function overrides `waitKey` to `True`, ensuring that as a user, we have the opportunity to inspect all debugging images for this function and can press a key when we are ready.



**Figure 11.3.** The results of our Python and OpenCV-based ANPR localization pipeline. This sample is very suitable to pass on to be OCR'd with Tesseract.

After that key is pressed, we break out of our loop, ignoring other candidates. Finally, we return the 2-tuple consisting of our ROI and license plate contour to the caller.

The *bottom* result is encouraging because Tesseract OCR should be able to decipher the characters.

### 11.2.7 Defining Our Tesseract ANPR Options

Leading up to this point, we've used our knowledge of OpenCV's morphological operations and contour processing to find the plate *and* ensure we have a clean image to send through the Tesseract OCR engine.

It is now time to do just that. Shifting our focus to OCR, let's define the `build_tesseract_options` method:

---

```

124 def build_tesseract_options(self, psm=7):
125     # tell Tesseract to only OCR alphanumeric characters
126     alphanumeric = "ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789"
127     options = "-c tessedit_char_whitelist={}".format(alphanumeric)
128
129     # set the PSM mode
130     options += " --psm {}".format(psm)
131
132     # return the built options string
133     return options

```

---

Tesseract and its Python-bindings brother, PyTesseract, accept a range of configuration options. We're only concerned with two:

- **Page segmentation method (PSM):** Tesseract's setting indicating layout analysis of the document/image. There are 13 modes of operation (Chapter 11, “*Improving OCR Results with Tesseract Options*” of the *Intro to OCR Bundle*), but we will default to 7 — “treat the image as a single text line” — per the `psm` parameter default.
- **Whitelist:** A listing of characters (letters, digits, symbols) that Tesseract will consider (i.e., report in the OCR'd results). Each of our whitelist characters is listed in the `alphanumeric` variable (**Line 126**).

**Lines 127–130** concatenate both into a formatted string with these option parameters. Our options are returned to the caller via **Line 133**.

### 11.2.8 The Heart of the ANPR Implementation

Our final method brings all the components together in one centralized place, so our driver script can instantiate a `PyImageSearchANPR` object, and then make a single function call. Let's implement `find_and_ocr`:

---

```

135 def find_and_ocr(self, image, psm=7, clearBorder=False):
136     # initialize the license plate text
137     lpText = None
138
139     # convert the input image to grayscale, locate all candidate
140     # license plate regions in the image, and then process the
141     # candidates, leaving us with the *actual* license plate
142     gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
143     candidates = self.locate_license_plate_candidates(gray)
144     (lp, lpCnt) = self.locate_license_plate(gray, candidates,
145                                               clearBorder=clearBorder)
146
147     # only OCR the license plate if the license plate ROI is not
148     # empty
149     if lp is not None:
150         # OCR the license plate
151         options = self.build_tesseract_options(psm=psm)
152         lpText = pytesseract.image_to_string(lp, config=options)
153         self.debug_imshow("License Plate", lp)
154
155     # return a 2-tuple of the OCR'd license plate text along with
156     # the contour associated with the license plate region
157     return (lpText, lpCnt)

```

---

This method accepts three parameters:

- i. `image`: The three-channel color image of the rear (or front) of a car with a license plate tag
- ii. `psm`: The Tesseract page segmentation mode
- iii. `clearBorder`: The flag indicating whether we'd like to clean up contours touching the border of the license plate ROI

Given our function parameters, we now:

- Convert the input `image` to grayscale (**Line 142**)
- Determine our set of license plate candidates from our `gray` image via the method we previously defined (**Line 143**)

- Locate the license plate from the `candidates` resulting in our `lp` ROI (**Lines 144 and 145**)

Assuming we've found a suitable plate (i.e., `lp` is not `None`), we set our PyTesseract options and perform OCR via the `image_to_string` method (**Lines 149–152**).

Finally, **Line 157** returns a 2-tuple consisting of the OCR'd `lpText` and `lpCnt` contour.

Phew! You did it! Nice job implementing the `PyImageSearchANPR` class.

If you found that implementing this class was challenging to understand, then I would recommend you study Module 1 of the `PylImageSearch Gurus` course ([http://pyimg.co/gurus\[22\]](http://pyimg.co/gurus[22])), where you'll learn the basics of computer vision and image processing.

In our next section, we'll create a Python script that utilizes the `PyImageSearchANPR` class to perform ALPR/ANPR on input images.

### 11.2.9 Creating Our ANPR and OCR Driver Script

Now that our `PyImageSearchANPR` class is implemented, we can move on to creating a Python driver script that will:

- i. Load an input image from disk
- ii. Find the license plate in the input image
- iii. OCR the license plate
- iv. Display the ANPR result to our screen

Let's take a look in the project directory and find our driver file `ocr_license_plate.py`:

---

```
1 # import the necessary packages
2 from pyimagesearch.anpr import PyImageSearchANPR
3 from pyimagesearch.helpers import cleanup_text
4 from imutils import paths
5 import argparse
6 import imutils
7 import cv2
```

---

We have our imports, namely our custom `PyImageSearchANPR` class that we implemented in the previous section and subsections.

Let's familiarize ourselves with this script's command line arguments:

---

```

9  # construct the argument parser and parse the arguments
10 ap = argparse.ArgumentParser()
11 ap.add_argument("-i", "--input", required=True,
12     help="path to input directory of images")
13 ap.add_argument("-c", "--clear-border", type=int, default=-1,
14     help="whether or to clear border pixels before OCR'ing")
15 ap.add_argument("-p", "--psm", type=int, default=7,
16     help="default PSM mode for OCR'ing license plates")
17 ap.add_argument("-d", "--debug", type=int, default=-1,
18     help="whether or not to show additional visualizations")
19 args = vars(ap.parse_args())

```

---

Our command line arguments include:

- `--input`: The required path to the input directory of vehicle images.
- `--clear-border`: A flag indicating if we'll clean up the edges of our license plate ROI prior to passing it to Tesseract.
- `--psm`: Tesseract's page segmentation mode; a `7` indicates that Tesseract should only look for one line of text.
- `--debug`: A Boolean indicating whether we wish to display an intermediate image processing pipeline debugging images.

With our imports in place, text cleanup utility defined, and an understanding of our command line arguments, now it is time to recognize license plates automatically!

---

```

21 # initialize our ANPR class
22 anpr = PyImageSearchANPR(debug=args["debug"] > 0)
23
24 # grab all image paths in the input directory
25 imagePaths = sorted(list(paths.list_images(args["input"])))

```

---

First, we instantiate our `PyImageSearchANPR` object while passing our `--debug` flag (**Line 22**). We also go ahead and bring in all the `--input` image paths with `imutils`' `paths` module (**Line 25**).

We'll process each of our `imagePaths` in hopes of finding and OCR'ing each license plate successfully:

---

```

27 # loop over all image paths in the input directory
28 for imagePath in imagePaths:
29     # load the input image from disk and resize it
30     image = cv2.imread(imagePath)

```

---

```

31     image = imutils.resize(image, width=600)
32
33     # apply automatic license plate recognition
34     (lpText, lpCnt) = anpr.find_and_ocr(image, psm=args["psm"],
35         clearBorder=args["clear_border"] > 0)
36
37     # only continue if the license plate was successfully OCR'd
38     if lpText is not None and lpCnt is not None:
39         # fit a rotated bounding box to the license plate contour and
40         # draw the bounding box on the license plate
41         box = cv2.boxPoints(cv2.minAreaRect(lpCnt))
42         box = box.astype("int")
43         cv2.drawContours(image, [box], -1, (0, 255, 0), 2)
44
45         # compute a normal (unrotated) bounding box for the license
46         # plate and then draw the OCR'd license plate text on the
47         # image
48         (x, y, w, h) = cv2.boundingRect(lpCnt)
49         cv2.putText(image, cleanup_text(lpText), (x, y - 15),
50             cv2.FONT_HERSHEY_SIMPLEX, 0.75, (0, 255, 0), 2)
51
52     # show the output ANPR image
53     print("[INFO] {}".format(lpText))
54     cv2.imshow("Output ANPR", image)
55     cv2.waitKey(0)

```

---

Looping over our `imagePaths`, we load and resize the image (**Lines 27–31**).

A call to our `find_and_ocr` method — while passing the `image`, `--psm` mode, and `--clear-border` flag — primes our ANPR pipeline pump to spit out the resulting OCR'd text and license plate contour on the other end.

**You've just performed ALPR/ANPR in the driver script!** If you need to revisit this method, refer to the walkthrough in Section 11.2.8, bearing in mind that the bulk of the work is done in the class methods leading up to the `find_and_ocr` method.

Assuming that both `lpText` and `lpCnt` did not return as `None` (**Line 38**), let's annotate the original input image with the OCR result. Inside the conditional, we:

- Calculate and draw the bounding box of the license plate contour (**Lines 41–43**)
- Annotate the cleaned up `lpText` string (**Lines 48–50**)
- Display the license plate string in the terminal and the annotated image in a GUI window (**Lines 53 and 54**)

You can now cycle through all of your `--input` directory images by pressing any key (**Line 55**).

You did it! Pat yourself on the back before proceeding to the results section — you deserve it.

### 11.2.10 Automatic License Plate Recognition Results

We are now ready to apply ALPR/ANPR using OpenCV and Python.

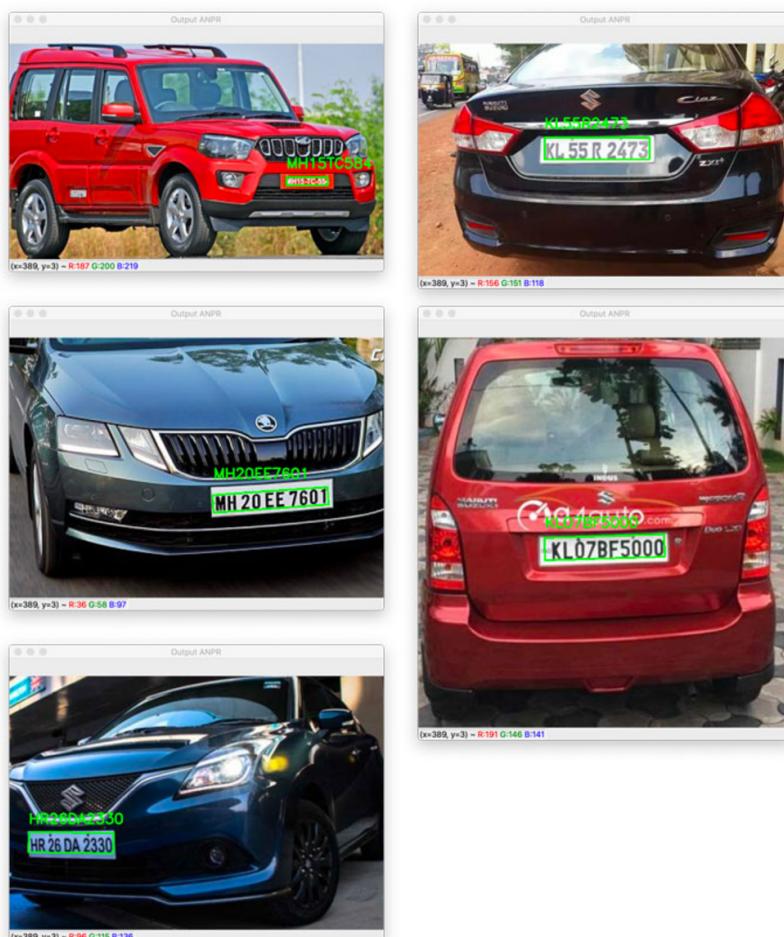
Open a terminal and execute the following command to review the ANPR results on our first set of sample images:

---

```
$ python ocr_license_plate.py --input license_plates/group1
[INFO] MH15TC584
[INFO] KL55R2473
[INFO] MH20EE7601
[INFO] KLO7BF5000
[INFO] HR26DA2330
```

---

As you can see from Figure 11.4, we've successfully applied ANPR to all of these images, including license/number plate examples on the vehicle's front or back.



**Figure 11.4.** Our ALPR/ANPR algorithm developed is successful on all five of the test images in the first group!

Let's try another set of images, this time where our ANPR solution doesn't work as well:

```
$ python ocr_license_plate.py --input license_plates/group2
[INFO] MHOZDW8351
[INFO] SICAL
[INFO] WMTA
```

While the first result image has the correct ANPR result, the other two are wildly incorrect (Figure 11.5, *left*).



**Figure 11.5.** *Left:* Unfortunately, “group 2” vehicle images lead to mixed results. In this case, we are \*not\* invoking the option to clear foreground pixels around the border of the license plate, which is detrimental to Tesseract’s ability to decipher the number plate. *Right:* By applying the `clear_border` option to “group 2” vehicle images, we see an improvement in the results. However, we still have some OCR mistakes.

The solution here is to apply our `clear_border` function to strip foreground pixels that touch the border of the image that confuse Tesseract OCR:

---

```
$ python ocr_license_plate.py --input license_plates/group2 --clear-border 1
[INFO] MHOZDW8351
[INFO] KA297999
[INFO] KE53E964
```

---

As Figure 11.5 (*right*) shows, we're able to improve the ANPR OCR results for these images by applying the `clear_border` function. However, there is still one mistake in each example. In the *top-right* case, the letter "Z" is mistaken for the digit "7." In the *bottom* case, the letter "L" is mistaken for the letter "E."

Although these are understandable mistakes, we would hope to do better. While our system is a great start (and is sure to impress our friends and family!), there are some obvious limitations and drawbacks associated with today's proof of concept. Let's discuss them, along with a few ideas for improvement.

### 11.2.11 Limitations and Drawbacks

As the previous section's ANPR results showed, our ANPR system sometimes worked well and other times it did not. Furthermore, something as simple as clearing any foreground pixels that touch the input license plate's borders improved license plate OCR accuracy.

Why is that?

The simple answer here is that Tesseract's OCR engine can be a bit sensitive. Tesseract will work best when you provide it with neatly cleaned and pre-processed images.

However, in real-world implementations, you may not be able to guarantee clear images. Instead, your images may be grainy or low quality, or the driver of a given vehicle may have a special cover on their license plate to obfuscate the view of it, making ANPR *even more* challenging.

This method will work well in controlled conditions. Still, suppose you want to build a system that works in *uncontrolled* environments. In that case, you'll need to start replacing components (namely license plate localization, character segmentation, and character OCR) with more advanced machine learning and deep learning models.

## 11.3 Summary

In this chapter, you learned how to build a basic Automatic License/Number Plate Recognition system using OpenCV and Python.

Our ANPR method relied on basic computer vision and image processing techniques to localize a license plate in an image, including morphological operations, image gradients, thresholding, bitwise operations, and contours.

This method will work well in controlled, predictable environments — like when lighting conditions are uniform across input images and license plates are standardized (e.g., dark characters on a light license plate background).

However, suppose you are developing an ANPR system that *does not* have a controlled environment. In that case, you'll need to start inserting machine learning and/or deep learning to replace parts of our plate localization pipeline.

HOG + Linear SVM is a good starting point for plate localization if your input license plates have a viewing angle that doesn't change more than a few degrees. If you're working in an unconstrained environment where viewing angles can vary dramatically, then deep learning-based models (e.g., Faster R-CNN, SSDs, and YOLO) will likely obtain better accuracy.



## Chapter 12

# Multi-Column Table OCR

Perhaps one of the more challenging applications of optical character recognition (OCR) is how to successfully OCR multi-column data, such as spreadsheets, tables, etc.

On the surface, OCR'ing tables seems like it should be an easier problem, right? Given that the document has a *guaranteed structure* to it, shouldn't we leverage this *a priori* knowledge and then be able to OCR each column in the table?

In most cases, yes, that would be the case. But unfortunately, we have a few problems to address:

- i. Tesseract isn't very good at multi-column OCR, *especially* if your image is noisy
- ii. You may need first to *detect* the table in the image before you can OCR it
- iii. Your OCR engine (Tesseract, cloud-based, etc.) may correctly OCR the text but be unable to associate the text into columns/rows

So, while OCR'ing multi-column data may appear to be an easy task, it's far harder since we may need to be responsible for associating text into columns and rows — and as you'll see, that is indeed the most complex part of our implementation.

The good news is that while OCR'ing multi-column data is certainly more demanding than other OCR tasks, it's not a "hard problem," provided you bring the right algorithms and techniques to the project.

In this chapter, you'll learn some of my tips and tricks to OCR multi-column data, and most importantly, associate rows/columns of text together.

## 12.1 Chapter Learning Objectives

In this chapter, you will:

- i. Discover my technique for associating rows and columns together
- ii. Learn how to detect tables of text/data in an image
- iii. Extract the detected table from an image
- iv. OCR the text in the table
- v. Apply hierarchical agglomerative clustering (HAC) to associate rows and columns
- vi. Build a Pandas `DataFrame` from the OCR'd data

## 12.2 OCR'ing Multi-Column Data

In the first part of this chapter, we'll discuss our multi-column OCR algorithm's basic process. This is the *exact* algorithm I use when I need to OCR multi-column data. It serves as a great starting point, and I recommend using it whenever you need to OCR a table.

From there, we'll review the directory structure for our project. We'll also install any additional required Python packages for the chapter.

With our development environment fully configured, we can move on to our implementation. We'll spend most of the chapter here, covering our multi-column OCR algorithm's details and inner workings.

We'll wrap up the chapter by applying our Python implementation to:

- i. Detect a table of text in an image
- ii. Extract the table
- iii. OCR the table
- iv. Build a Pandas `DataFrame` from the table to process it, query it, etc.

### 12.2.1 Our Multi-Column OCR Algorithm

Our multi-column OCR algorithm is a multi-step process. To start, we need to accept an input image containing a table, spreadsheet, etc. (Figure 12.1, *left*). Given this image, we then need to extract the table itself (*right*).



**Figure 12.1.** Left: Our input image containing statistics from the back of a Michael Jordan baseball card (yes, baseball — that's not a typo) [43]. Right: Our goal is to detect and extract the table of data from the input image.

Once we have the table, we can apply OCR and text localization to generate the  $(x, y)$ -coordinates for the text bounding boxes. **It's paramount that we obtain these bounding box coordinates.**

To associate columns of text together, we need to group pieces of text based on their starting  $x$ -coordinate.

Why starting  $x$ -coordinate? Well, keep in mind the structure of a table, spreadsheet, etc. Each column's text will have near-identical starting  $x$ -coordinates because they belong to the *same* column (take a second now to convince yourself that the statement is true).

We can thus exploit that knowledge and then cluster groups of text together with near-identical  $x$ -coordinates.

But the question remains, *how do we do the actual grouping?*

The answer is to use a special type of clustering algorithm called hierarchical agglomerative clustering (HAC). If you've ever taken a data science, machine learning, or text mining class before, then you've likely encountered HAC before.

A full review of the HAC algorithm is outside the scope of this chapter. Still, the general idea is that we take a “bottom-up” approach, starting with our initial data points (i.e., the  $x$ -coordinates of the text bounding boxes) as individual clusters, each containing only one observation.

We then compute the distance between each of the coordinates and start grouping observations with a distance  $< T$ , where  $T$  is a pre-defined threshold.

At each iteration of HAC, we choose the two clusters with the smallest distance and merge them, again, provided that the distance threshold requirement is met.

We continue clustering until no other clusters can be formed, typically due to no two clusters falling within our threshold requirement.

In the context of multi-column OCR, applying HAC to the  $x$ -coordinate value results in clusters that have identical or near-identical  $x$ -coordinates. And since we assume that text belonging to the same column will have similar/near-identical  $x$ -coordinates, we can associate columns together.

While this technique to multi-column OCR may seem complicated, it's straightforward, especially since we'll be able to utilize the `AgglomerativeClustering` implementation in the scikit-learn library (<http://pyimg.co/y3bic> [44]).

That said, implementing agglomerative clustering by hand is a good exercise if you are interested in learning more about machine learning and data science techniques. I've had to implement HAC 3–4 times before in my career, predominately when I was in college.

If you want to learn more about HAC, I recommend the scikit-learn documentation (<http://pyimg.co/y3bic> [44]) and the excellent article by Cory Maklin (<http://pyimg.co/glaxm> [45]).

For a more theoretical and mathematically motivated treatment of agglomerative clustering, including clustering algorithms in general, I highly recommend *Data Mining: Practical Machine Learning Tools and Techniques* by Witten et al. [46].

### 12.2.2 Project Structure

Let's get started by reviewing our project directory structure:

---

```
|-- michael_jordan_stats.png
|-- multi_column_ocr.py
|-- results.csv
```

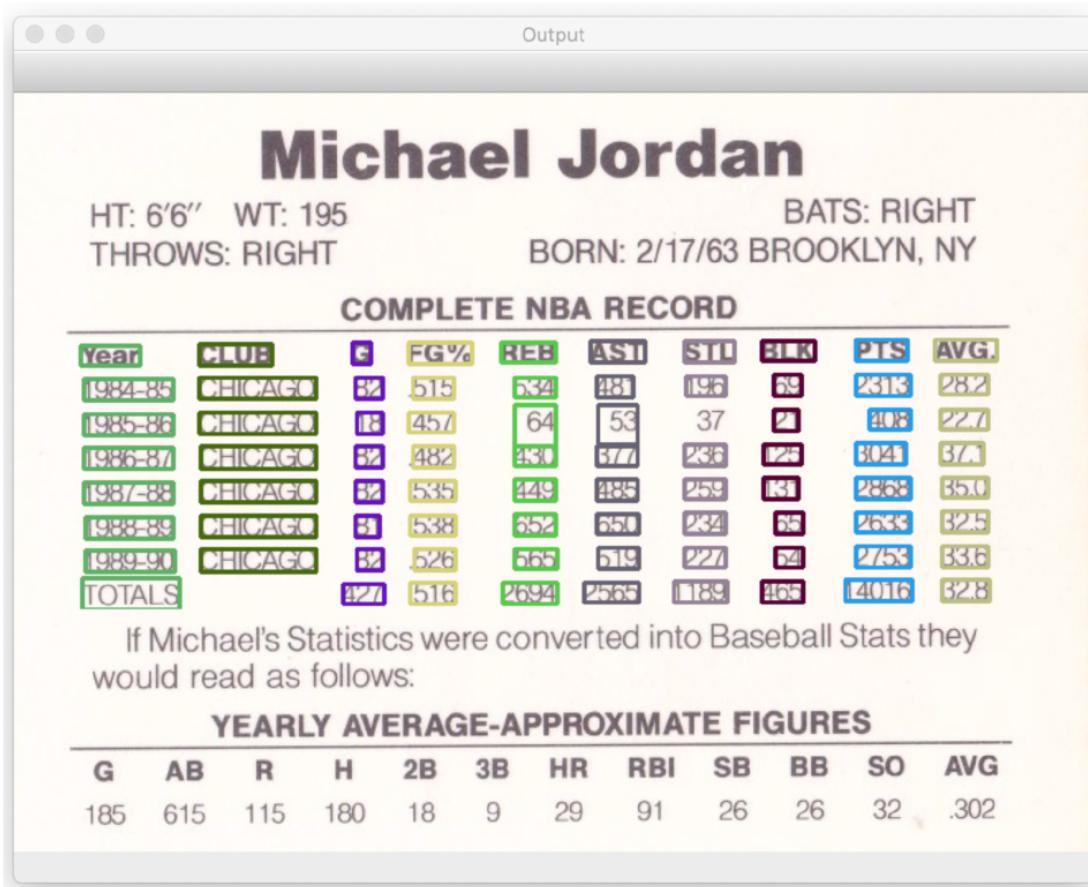
---

As you can see, our project directory structure is quite simple — but don't let the simplicity fool you!

Our `multi_column_ocr.py` script will accept an input image, which is `michael_jordan_stats.png`, detect the table of data, extract it, and then OCR it, associating rows/columns along the way.

For reference, our example image is a scan of the Michael Jordan baseball card (Figure 12.2), when he took a year off from basketball to play baseball after his father died.

Our Python script will be able to OCR the table, parse out his stats, and then output them as OCR'd text as a CSV file (`results.csv`).



**Figure 12.2.** Sample output of applying multi-column OCR. Each column is correctly detected, and each cell in a row is assigned to a particular column.

### 12.2.3 Installing Required Packages

Our Python script will display a nicely formatted table of OCR'd text to our terminal. Still, to generate this formatted table, we need to utilize the `tabulate` Python package ([\[47\]](http://pyimg.co/2r83u)).

You can install `tabulate` using the following command:

---

```
$ workon your_env_name # optional
$ pip install tabulate
```

---

If you are using Python virtual environments, don't forget to use the `workon` command (or

equivalent) to access your virtual environment *before* installing `tabulate` (otherwise, your `import` command will fail).

Again, the `tabulate` package is used for *display purposes only* and *does not* impact our actual multi-column OCR algorithm. If you do not wish to install `tabulate`, that's perfectly fine. You will just need to comment out the 2–3 lines of code that utilize it in our Python script.

#### 12.2.4 Implementing Multi-Column OCR

We are now ready to implement multi-column OCR! Open the `multi_column_ocr.py` file in your project directory structure, and let's get to work:

---

```

1 # import the necessary packages
2 from sklearn.cluster import AgglomerativeClustering
3 from pytesseract import Output
4 from tabulate import tabulate
5 import pandas as pd
6 import numpy as np
7 import pytesseract
8 import argparse
9 import imutils
10 import cv2

```

---

We start by importing our required Python packages. We have several packages we haven't (or at the very least, not often) worked with before, so let's review the important ones.

First, we have our `AgglomerativeClustering` implementation from scikit-learn. As I discussed in Section 12.2.1, we'll use HAC to cluster text together into columns. This implementation will allow us to do just that.

The `tabulate` package (mentioned in Section 12.2.3) will allow us to print a nicely formatted table of data to our terminal after OCR has been performed. This package is optional, so if you don't want to install it, simply comment out the `import` line and **Line 178** later in our implementation.

We then have the `pandas` library, which is super common amongst data scientists. We'll use `pandas` in this chapter for its ability to construct and manage tabular data easily.

The rest of the imports should look familiar to you as we have used each of them many times throughout this text.

Let's move on to our command line arguments:

---

```

12 # construct the argument parser and parse the arguments
13 ap = argparse.ArgumentParser()

```

---

```

14 ap.add_argument("-i", "--image", required=True,
15     help="path to input image to be OCR'd")
16 ap.add_argument("-o", "--output", required=True,
17     help="path to output CSV file")
18 ap.add_argument("-c", "--min-conf", type=int, default=0,
19     help="minimum confidence value to filter weak text detection")
20 ap.add_argument("-d", "--dist-thresh", type=float, default=25.0,
21     help="distance threshold cutoff for clustering")
22 ap.add_argument("-s", "--min-size", type=int, default=2,
23     help="minimum cluster size (i.e., # of entries in column)")
24 args = vars(ap.parse_args())

```

---

As you can see, we have several command line arguments. Let's discuss each of them:

- `--image`: The path to the input image containing the table, spreadsheet, etc. that we want to detect and OCR
- `--output`: Path to the output CSV file that will contain the column data we extracted
- `--min-conf`: Used to filter out weak text detections
- `--dist-thresh`: Distance threshold cutoff (in pixels) for when applying HAC; you may need to tune this value for your images and datasets
- `--min-size`: Minimum number of data points in a cluster for it to be considered a column

The most important command line arguments here are `--dist-thresh` and `--min-size`.

When applying HAC, we need to use a distance threshold cutoff. If you allow clustering to continue indefinitely, HAC will continue to cluster at each iteration until you end up, trivially, with one cluster that contains *all* data points.

Instead, you apply a distance threshold to *stop the clustering process* when no two clusters have a distance less than the threshold.

For your purposes, you'll need to examine the image data with which you are working. If your tabular data have large amounts of whitespace between each row, then you'll likely need to *increase* the `--dist-thresh` (Figure 12.3, *left*).

Otherwise, if there is less whitespace between each row, then the `--dist-thresh` could *decrease* accordingly (Figure 12.3, *right*).

**Left Table Data:**

G	FG%
82	0.515
18	0.457
82	0.482
82	0.535
81	0.526
82	0.516

**Right Table Data:**

G	FG%
82	0.515
18	0.457
82	0.482
82	0.535
81	0.526
82	0.516

**Figure 12.3.** Left: The *larger* the distance between text in cells, the *larger* your `--dist-thresh` should be. Right: The *smaller* the distance between text in cells, the *smaller* the `--dist-thresh` can be.

**Setting your `--dist-thresh` properly is *paramount* to OCR’ing multi-column data, be sure to experiment with different values.**

The `--min-size` command line argument is also important. At each iteration of our clustering algorithm, HAC will examine two clusters, each of which could contain *multiple* data points or just a *single* data point. If the distance between the two clusters is less than the `--dist-thresh`, HAC will merge them.

However, there will always be outliers, pieces of text that far outside the table, or just noise in the image. If Tesseract detects this text, then HAC will try to cluster it. But is there a way to prevent these clusters from being reported in our results?

A simple way is to check the number of text data points inside a given cluster after HAC is complete.

In this case, we set `--min-size=2`, meaning that if a cluster has  $\leq 2$  data points inside of it, we consider it an outlier and ignore it. You may need to tweak this variable for your applications, but I suggest tuning `--dist-thresh` first.

With our command line arguments taken care of, let's start our image processing pipeline:

---

```

26 # set a seed for our random number generator
27 np.random.seed(42)
28
29 # load the input image and convert it to grayscale
30 image = cv2.imread(args["image"])
31 gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

```

---

**Line 27** sets the seed of our pseudorandom number generator. We do this so we can generate colors for each column of text we detect (useful for visualization purposes).

We then load our input `--image` from disk and convert it to grayscale.

Our next code block detects large blocks of text in our `image`, taking a similar process to our chapter on OCR'ing passports in the “*Intro to OCR*” *Bundle*:

---

```

33 # initialize a rectangular kernel that is ~5x wider than it is tall,
34 # then smooth the image using a 3x3 Gaussian blur and then apply a
35 # blackhat morphological operator to find dark regions on a light
36 # background
37 kernel = cv2.getStructuringElement(cv2.MORPH_RECT, (51, 11))
38 gray = cv2.GaussianBlur(gray, (3, 3), 0)
39 blackhat = cv2.morphologyEx(gray, cv2.MORPH_BLACKHAT, kernel)
40
41 # compute the Scharr gradient of the blackhat image and scale the
42 # result into the range [0, 255]
43 grad = cv2.Sobel(blackhat, ddepth=cv2.CV_32F, dx=1, dy=0, ksize=-1)
44 grad = np.absolute(grad)
45 (minVal, maxVal) = (np.min(grad), np.max(grad))
46 grad = (grad - minVal) / (maxVal - minVal)
47 grad = (grad * 255).astype("uint8")
48
49 # apply a closing operation using the rectangular kernel to close
50 # gaps in between characters, apply Otsu's thresholding method, and
51 # finally a dilation operation to enlarge foreground regions
52 grad = cv2.morphologyEx(grad, cv2.MORPH_CLOSE, kernel)
53 thresh = cv2.threshold(grad, 0, 255,
54     cv2.THRESH_BINARY | cv2.THRESH_OTSU)[1]
55 thresh = cv2.dilate(thresh, None, iterations=3)
56 cv2.imshow("Thresh", thresh)

```

---

First, we construct a large rectangular kernel that is  $\approx 5x$  wider than it is tall (**Line 37**). We then apply a small Gaussian blur to the image and then compute the blackhat output, revealing dark regions on a light background (i.e., dark text on a light background).

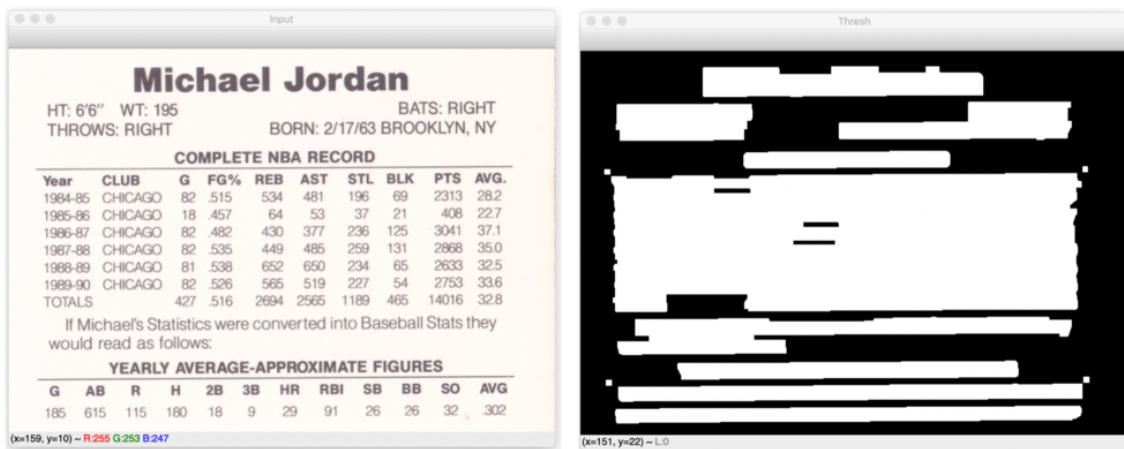
The next step is to compute the gradient magnitude representation of the blackhat image and scale the output image back to the range  $[0, 255]$  (**Lines 43–47**).

With the gradient image computed, we can now apply a closing operation ([Line 52](#)). We use our large rectangular kernel here to close gaps between rows of text in the table.

We finish the pipeline by thresholding the image and then applying a series of dilations to enlarge foreground regions.

Figure 12.4 displays the output of this pre-processing pipeline. On the *left*, we have our original input image. This image is the back of a Michael Jordan baseball card (when he left the NBA for a year to play baseball). Since they didn't have any baseball stats for Jordan, they included his basketball stats on the back of the card, despite being a baseball card (weird, I know, which is also why his baseball cards are collector's items).

**Our goal is to OCR his “Complete NBA Record” table.** And if you look at Figure 12.4 (*right*), you can see that the table is visible as one big rectangular-like blob.



**Figure 12.4.** *Left:* Our input image contains a table of data that we want to apply multi-column OCR to. *Right:* Output of applying a closing morphological operation to detect large blocks of text. The largest foreground region is the “Complete NBA Record” table we’re interested in and want to OCR.

Our next code block handles detecting and extracting this table:

```

58 # find contours in the thresholded image and grab the largest one,
59 # which we will assume is the stats table
60 cnts = cv2.findContours(thresh.copy(), cv2.RETR_EXTERNAL,
61     cv2.CHAIN_APPROX_SIMPLE)
62 cnts = imutils.grab_contours(cnts)
63 tableCnt = max(cnts, key=cv2.contourArea)
64
65 # compute the bounding box coordinates of the stats table and extract
66 # the table from the input image

```

```

67  (x, y, w, h) = cv2.boundingRect(tableCnt)
68  table = image[y:y + h, x:x + w]
69
70 # show the original input image and extracted table to our screen
71 cv2.imshow("Input", image)
72 cv2.imshow("Table", table)

```

---

First, we detect all contours in our thresholded image (**Lines 60–62**). We then find the single largest contour based on the area of the contour (**Line 63**).

We're assuming that the single largest contour is our table, and as we can verify from Figure 12.4 (right), that is indeed the case. However, you may need to update the contour processing to find the table using the previous chapters' methods for your projects. The method covered here is certainly *not* a one-size-fits-all solution.

Given the contour corresponding to the table, `tableCnt`, we compute its bounding box and extract the `table` itself (**Lines 67 and 68**).

The extracted `table` is then displayed to our screen on **Line 72** (Figure 12.5).

Year	CLUB	G	FG%	REB	AST	STL	BLK	PTS	AVG.
1984-85	CHICAGO	82	.515	534	481	196	69	2313	28.2
1985-86	CHICAGO	18	.457	64	53	37	21	408	22.7
1986-87	CHICAGO	82	.482	430	377	236	125	3041	37.1
1987-88	CHICAGO	82	.535	449	485	259	131	2868	35.0
1988-89	CHICAGO	81	.538	652	650	234	65	2633	32.5
1989-90	CHICAGO	82	.526	565	519	227	54	2753	33.6
TOTALS		427	.516	2694	2565	1189	465	14016	32.8

(x=15, y=38) ~ R:255 G:254 B:251

**Figure 12.5.** Using image processing techniques, we can automatically detect the largest table present inside our image.

Now that we have our statistics `table`, let's OCR it:

```

74 # set the PSM mode to detect sparse text, and then localize text in
75 # the table
76 options = "--psm 6"
77 results = pytesseract.image_to_data(
78     cv2.cvtColor(table, cv2.COLOR_BGR2RGB),
79     config=options,
80     output_type=Output.DICT)

```

---

```

81
82 # initialize a list to store the (x, y)-coordinates of the detected
83 # text along with the OCR'd text itself
84 coords = []
85 ocrText = []

```

---

**Lines 76–80** use Tesseract to compute bounding box locations for each piece of text in the table.

Additionally, notice how we are using `--psm 6` here, the reason being that this particular page segmentation mode works well when the text is a single uniform block of text.

Most tables of data *are* going to be uniform. They will leverage near-identical font and font sizes.

If `--psm 6` is not working well for you, you should try the other PSM modes covered in Chapter 11 (*Improving OCR Results with Tesseract Options*) of the "*Intro to OCR*" Bundle. Specifically, I suggest taking a look at `--psm 11` on detecting sparse text — that mode could work well for tabular data as well.

After applying OCR text detection, we then initialize two lists: `coords` and `ocrText`. The `coords` list will store the  $(x, y)$ -coordinates of the text bounding boxes, while `ocrText` will store the actual OCR'd text itself.

Let's move on to looping over each of our text detections:

---

```

87 # loop over each of the individual text localizations
88 for i in range(0, len(results["text"])):
89     # extract the bounding box coordinates of the text region from
90     # the current result
91     x = results["left"][i]
92     y = results["top"][i]
93     w = results["width"][i]
94     h = results["height"][i]
95
96     # extract the OCR text itself along with the confidence of the
97     # text localization
98     text = results["text"][i]
99     conf = int(results["conf"][i])
100
101    # filter out weak confidence text localizations
102    if conf > args["min_conf"]:
103        # update our text bounding box coordinates and OCR'd text,
104        # respectively
105        coords.append((x, y, w, h))
106        ocrText.append(text)

```

---

**Lines 91–94** extract the bounding box coordinates from the detected text region, while **Lines 98 and 99** extract the OCR'd text itself, along with the confidence of the text detection.

**Line 102** filters out weak text detections. If the `conf` is less than our `--min-conf`, we ignore the text detection. Otherwise, we update our `coords` and `ocrText` lists, respectively.

We can now move on to the clustering phase of the project:

---

```

108 # extract all x-coordinates from the text bounding boxes, setting the
109 # y-coordinate value to zero
110 xCoords = [(c[0], 0) for c in coords]
111
112 # apply hierarchical agglomerative clustering to the coordinates
113 clustering = AgglomerativeClustering(
114     n_clusters=None,
115     affinity="manhattan",
116     linkage="complete",
117     distance_threshold=args["dist_thresh"])
118 clustering.fit(xCoords)
119
120 # initialize our list of sorted clusters
121 sortedClusters = []

```

---

**Line 110** extracts all *x*-coordinates from the text bounding boxes. Notice how we are forming a proper  $(x, y)$ -coordinate tuple here, but setting  $y = 0$  for each entry. Why is  $y$  set to 0?

The answer is two-fold:

- i. First, to apply HAC, we need a set of input vectors (also called “feature vectors”). Our input vectors must be *at least* 2-d, so we add a trivial dimension containing a value of 0.
- ii. Secondly, we aren’t interested in the *y*-coordinate value. We only want to cluster on the *x*-coordinate positions. Pieces of text with similar *x*-coordinates are likely to be part of a column in a table.

From there, **Lines 113–118** apply hierarchical agglomerative clustering. We set `n_clusters` to `None` since we *do not know* how many clusters we want to generate — we instead wish HAC to form clusters naturally and continue creating clusters using our `distance_threshold`. Once no two clusters have a distance less than `--dist-thresh`, we stop the clustering processing.

Also, note that we are using the Manhattan distance function here. Why not some other distance function, such as Euclidean?

While you can (and should) experiment with our distance metrics, Manhattan tends to be an appropriate choice here. We want to be very stringent on our requirement that *x*-coordinates lie close together. But again, I suggest you experiment with other distance methods.

For more details on scikit-learn's AgglomerativeClustering implementation, including a full review of the parameters and arguments, be sure to refer to scikit-learn's documentation on the method: (<http://pyimg.co/y3bic> [44]).

Now that our clustering is complete, let's loop over each of the unique clusters:

---

```

123 # loop over all clusters
124 for l in np.unique(clustering.labels_):
125     # extract the indexes for the coordinates belonging to the
126     # current cluster
127     idxs = np.where(clustering.labels_ == l)[0]
128
129     # verify that the cluster is sufficiently large
130     if len(idxs) > args["min_size"]:
131         # compute the average x-coordinate value of the cluster and
132         # update our clusters list with the current label and the
133         # average x-coordinate
134         avg = np.average([coords[i][0] for i in idxs])
135         sortedClusters.append((l, avg))
136
137 # sort the clusters by their average x-coordinate and initialize our
138 # data frame
139 sortedClusters.sort(key=lambda x: x[1])
140 df = pd.DataFrame()

```

---

**Line 124** loops through all unique cluster labels. We then use NumPy to find the indexes of all data points with the current label, `l` (**Line 127**), thus implying they all belong to the same cluster.

**Line 130** then verifies that the current cluster has more than `--min-size` items inside of it.

We then compute the average of the x-coordinates inside the cluster and update our `sortedClusters` list with a 2-tuple containing the current label, `l`, along with the average x-coordinate value.

**Line 139** sorts our `sortedClusters` list based on our average x-coordinate. We perform this sorting operation such that our clusters are sorted *left-to-right* across the page.

Finally, we initialize an empty `DataFrame` to store our multi-column OCR results.

Let's now loop over our sorted clusters:

---

```

142 # loop over the clusters again, this time in sorted order
143 for (l, _) in sortedClusters:
144     # extract the indexes for the coordinates belonging to the
145     # current cluster
146     idxs = np.where(clustering.labels_ == l)[0]
147

```

---

---

```

148     # extract the y-coordinates from the elements in the current
149     # cluster, then sort them from top-to-bottom
150     yCoords = [coords[i][1] for i in idxs]
151     sortedIdxs = idxs[np.argsort(yCoords)]
152
153     # generate a random color for the cluster
154     color = np.random.randint(0, 255, size=(3,), dtype="int")
155     color = [int(c) for c in color]

```

---

**Line 143** loops over the labels in our `sortedClusters` list. We then find the indexes (`idxs`) of all data points belonging to the current cluster label, `l`.

Using the indexes, we extract the *y*-coordinates from all pieces of text in the cluster and sort them from *top-to-bottom* (**Lines 150 and 151**).

We also initialize a random `color` for the current column (so we can visualize which pieces of text belong to which column).

Let's loop over each of the pieces of text in the column now:

---

```

157     # loop over the sorted indexes
158     for i in sortedIdxs:
159         # extract the text bounding box coordinates and draw the
160         # bounding box surrounding the current element
161         (x, y, w, h) = coords[i]
162         cv2.rectangle(table, (x, y), (x + w, y + h), color, 2)
163
164     # extract the OCR'd text for the current column, then construct
165     # a data frame for the data where the first entry in our column
166     # serves as the header
167     cols = [ocrText[i].strip() for i in sortedIdxs]
168     currentDF = pd.DataFrame({cols[0]: cols[1:]})
169
170     # concatenate *original* data frame with the *current* data
171     # frame (we do this to handle columns that may have a varying
172     # number of rows)
173     df = pd.concat([df, currentDF], axis=1)

```

---

For each `sortedIdx`, we grab the bounding box (*x*, *y*)-coordinates for the piece of text and draw it on our `table` (**Lines 158–162**).

We then grab all pieces of `ocrText` in the current cluster, sorted from *top-to-bottom* (**Line 167**). **These pieces of text represent one unique column (`cols`) in the table.**

Now that we've extracted the current column, we create a separate `DataFrame` for it, assuming that the first entry in the column is the header and the rest is the data (**Line 168**).

Finally, we concatenate our original data frame, `df`, with our new data frame, `currentDF` (**Line 173**). We perform this concatenation operation to handle cases where some columns

will have more rows than others (either naturally, due to table design, or due to the Tesseract OCR engine missing a piece of text in a row).

At this point our table OCR process is complete, we just need to save the table to disk:

---

```

175 # replace NaN values with an empty string and then show a nicely
176 # formatted version of our multi-column OCR'd text
177 df.fillna("", inplace=True)
178 print(tabulate(df, headers="keys", tablefmt="psql"))
179
180 # write our table to disk as a CSV file
181 print("[INFO] saving CSV file to disk...")
182 df.to_csv(args["output"], index=False)
183
184 # show the output image after performing multi-column OCR
185 cv2.imshow("Output", image)
186 cv2.waitKey(0)

```

---

If some columns have more entries than others, then the empty entries will be filled with a “Not a Number” (NaN) value. We replace all NaN values with an empty string on **Line 177**.

**Line 178** displays a nicely formatted table to our terminal, demonstrating that we’ve OCR’d the multi-column data successfully.

We then save our data frame to disk as a CSV file on **Line 182**.

Finally, we display our output image to our screen on **Line 185**.

### 12.2.5 Multi-Column OCR Results

We are now ready to see our multi-column OCR script in action!

Open up a terminal and execute the following command:

---

```
$ python multi_column_ocr.py --image michael_jordan_stats.png --output results.csv
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
|     | Year    | CLUB   | G      | FG%    | REB    | AST    | STL    | BLK    | PTS    | AVG.   |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 0  | 1984-85 | CHICAGO | 82    | 515    | 534    | 481    | 196    | 69     | 2313   | 282    |
| 1  | 1985-86 | CHICAGO | 18    | 0.457  | 64     | 53     | ar     | A      | 408    | 227    |
| 2  | 1986-87 | CHICAGO | 82    | 482    | 430    | 377    | 236    | 125    | 3041   | 37.1   |
| 3  | 1987-88 | CHICAGO | 82    | 535    | 449    | 485    | 259    | 131    | 2868   | 35     |
| 4  | 1988-89 | CHICAGO | 81    | 538    | 652    | 650    | 234    | 65     | 2633   | 325    |
| 5  | 1989-90 | CHICAGO | 82    | 526    | 565    | 519    | 227    | 54     | 2763   | 33.6   |
| 6  | TOTALS  |          | 427   | 516    | 2694   | 2565   | 1189   | 465    | 14016  | 328    |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
[INFO] saving CSV file to disk...
```

---

Figure 12.6 (top) displays the extracted table. We then apply hierarchical agglomerative clustering (HAC) to OCR the table, resulting in the *bottom* figure.

Year	CLUB	G	FG%	REB	AST	STL	BLK	PTS	Avg.
1984-85	CHICAGO	82	.515	534	481	196	69	2313	28.2
1985-86	CHICAGO	18	.457	64	53	37	21	408	22.7
1986-87	CHICAGO	82	.482	430	377	236	125	3041	37.1
1987-88	CHICAGO	82	.535	449	485	259	131	2868	35.0
1988-89	CHICAGO	81	.538	652	650	234	65	2633	32.5
1989-90	CHICAGO	82	.526	565	519	227	54	2753	33.6
TOTALS		427	.516	2694	2565	1189	465	14016	32.8

(x=15, y=38) ~ R:255 G:254 B:251

Output

# Michael Jordan

HT: 6'6" WT: 195      BATS: RIGHT  
THROWS: RIGHT      BORN: 2/17/63 BROOKLYN, NY

**COMPLETE NBA RECORD**

Year	CLUB	G	FG%	REB	AST	STL	BLK	PTS	Avg.
1984-85	CHICAGO	82	.515	534	481	196	69	2313	28.2
1985-86	CHICAGO	18	.457	64	53	37	21	408	22.7
1986-87	CHICAGO	82	.482	430	377	236	125	3041	37.1
1987-88	CHICAGO	82	.535	449	485	259	131	2868	35.0
1988-89	CHICAGO	81	.538	652	650	234	65	2633	32.5
1989-90	CHICAGO	82	.526	565	519	227	54	2753	33.6
TOTALS		427	.516	2694	2565	1189	465	14016	32.8

If Michael's Statistics were converted into Baseball Stats they would read as follows:

**YEARLY AVERAGE-APPROXIMATE FIGURES**

G	AB	R	H	2B	3B	HR	RBI	SB	BB	SO	Avg
185	615	115	180	18	9	29	91	26	26	32	.302

**Figure 12.6.** Top: The input table of data that we want to apply multi-column OCR to. Bottom: The output of applying multi-column OCR. Text bounding boxes with the *same color* belong to the *same column*, demonstrating that our algorithm can associate text with specific columns.

**Notice how we have color-coded the columns of text.** Text with the same bounding box color belongs to the *same column*, demonstrating that our HAC method successfully associated columns of text together.

A text version of our OCR'd table can be seen in the terminal output above. Our OCR results are very accurate for the most part, but there are a few notable issues to call out.

To start, the field goal percentage (`FG%`) is missing the decimal spot for all but one row, likely because the image was too low resolution and Tesseract could not successfully detect the decimal point. Luckily, that is an easy fix — use basic string parsing/regular expressions to insert the decimal point *or* just cast the string to a `float` data type. The decimal point will be added automatically.

The same issue can be found in the average points per game (`AVG.`) column — again, the decimal point is missing.

This one is slightly harder to resolve, though. If we were to cast to a `float` simply, then the text `282` would be incorrectly parsed as `0.282`. Instead, what we can do is:

- i. Check the length of the string
- ii. If the string has *four characters*, then the decimal point has already been added, so no further work is required
- iii. Otherwise, the string has *three characters*, so insert a decimal point between the second and third characters

As I've said many times throughout this book, any time, you can leverage any *a priori* knowledge regarding the OCR task at hand, the *far easier* time you'll have. In this case, our domain knowledge tells us which columns should have decimal points, so we can leverage text post-processing heuristics to improve our OCR results further, even when Tesseract performs less than optimally.

The most significant issues with our OCR results can be found in the `STL` and `BLK` columns, where the OCR results are simply incorrect. There's not much we can do about that since that is a problem with Tesseract's results and *not* our column-grouping algorithm.

You can follow any of the suggestions throughout this book to improve your Tesseract OCR accuracy. Since this chapter focuses *specifically* on OCR'ing multi-column data, and most importantly, *the algorithm powering it*, we're not going to spend a ton of time focusing on improvements to Tesseract options and configurations here.

After our Python script has been executed, we have an output `results.csv` file containing our table serialized to disk as a CSV file. Let's take a look at its contents:

---

```
$ cat results.csv
Year,CLUB,G,FG%,REB,AST,STL,BLK,PTS,Avg.
1984-85,CHICAGO,82,515,534,481,196,69,2313,282
1985-86,CHICAGO,18,.457,64,53,ar,A,408,227
1986-87,CHICAGO,82,482,430,377,236,125,3041,37.1
1987-88,CHICAGO,82,535,449,485,259,131,2868,35.0
1988-89,CHICAGO,81,538,652,650,234,65,2633,325
1989-90,CHICAGO,82,526,565,519,227,54,2763,33.6
TOTALS,,427,516,2694,2565,1189,465,14016,328
```

---

As you can see, our OCR'd table has been written to disk as a CSV file. You can take this CSV file and further process it using data mining techniques, etc.

## 12.3 Summary

In this chapter, you learned how to OCR multi-column data using the Tesseract OCR engine and hierarchical agglomerative clustering (HAC).

Our multi-column OCR algorithm works by:

- i. Detecting tables of text in an input image using gradients and morphological operations
- ii. Extracting the detected table
- iii. Using Tesseract (or equivalent) to localize text in the table and extract the bounding box ( $x, y$ )-coordinates of the text in the table
- iv. Applying HAC to cluster on the  $x$ -coordinate of the table with a maximum distance threshold cutoff

Essentially what we are doing here is grouping text localizations together that have  $x$ -coordinates that are either *identical* or are *very close* to each other.

Why does this method work?

Well, consider the structure of a spreadsheet, table, or any other document that has multiple columns. In that case, the data in each column will have near identical starting  $x$ -coordinates because they belong to the *same* column. We can thus exploit that knowledge and then cluster groups of text together with near-identical  $x$ -coordinates.

While this method is simplistic, it works quite well in practice.



## Chapter 13

# Blur Detection in Text and Documents

Whenever I teach computer vision to beginners, one of the most important points I try to drive home early on is that **it's far easier to write code to accurately process images that are captured in *ideal conditions* (appropriate lighting, high contrast, etc.) rather than *noisy conditions* (low lighting, blur, etc.).**

For example, suppose we are trying to build a pipeline to OCR video streams (which we'll do in the next chapter). In that case, we'll inevitably have frames in the video stream that are low quality, typically due to motion blur or the camera lens refocusing itself. Those frames will more than likely be *impossible* to OCR.

Instead of attempting to OCR these low-quality frames, which would more than likely result in erroneous OCR output, **what if we could *detect* and *discard* such frames?**

In this chapter, you'll learn how to detect low-quality images that will be extremely challenging to OCR. This detection method could be used in a user feedback-based application where readers are capturing photos of bank checks (for mobile deposits), invoices, forms, etc. If a user captures a low-quality frame, instead of OCR'ing it, you can instead report back to the user to capture a better version of the image (and provide helpful tips to improve photo quality, of course).

In the next chapter, you'll learn how to use the techniques covered here to detect blurry, low-quality frames from a video stream, discard them, and *only* OCR the high-quality frames.

The next two chapters' contents have critical applications in real-world OCR scenarios, so be sure to spend extra time reviewing them!

### 13.1 Chapter Learning Objectives

In this chapter, you will:

- Discover why we wish to detect blur in images before attempting to OCR them
- Learn about the fast Fourier transform (FFT)
- Gain an understanding of how the FFT can be used to detect blurry text
- Implement the FFT for blur detection using OpenCV and Python

## 13.2 Detecting Blurry Text and Documents

Back in 2015, I authored a tutorial on the PyImageSearch blog entitled *Blur Detection with OpenCV* (<http://pyimg.co/i68y> [48]). This method was based on the variance of the Laplacian operator. One of the primary benefits is that the variance of the Laplacian is *dead simple* to implement, requiring only a single line of code.

**However, the downside is that the Laplacian method required significant manual tuning to define the “threshold” at which an image was considered blurry or not.** If you could control your lighting conditions, environment, and image capturing process, it worked quite well — but if not, you would obtain mixed results.

The method we'll be covering in this chapter relies on computing the FFT of the image. It still requires some manual tuning, but as we'll find out, the FFT blur detector we'll be covering is *far more robust and reliable* than the variance of the Laplacian method.

This chapter will primarily focus on detecting blurry documents and text. In the next chapter, we'll extend this method to help us OCR video streams.

### 13.2.1 What Is Blur Detection and Why Do We Want to Detect Blur?

As the name suggests, blur detection is the process of detecting whether an image is blurry or not. Possible applications of blur detection include:

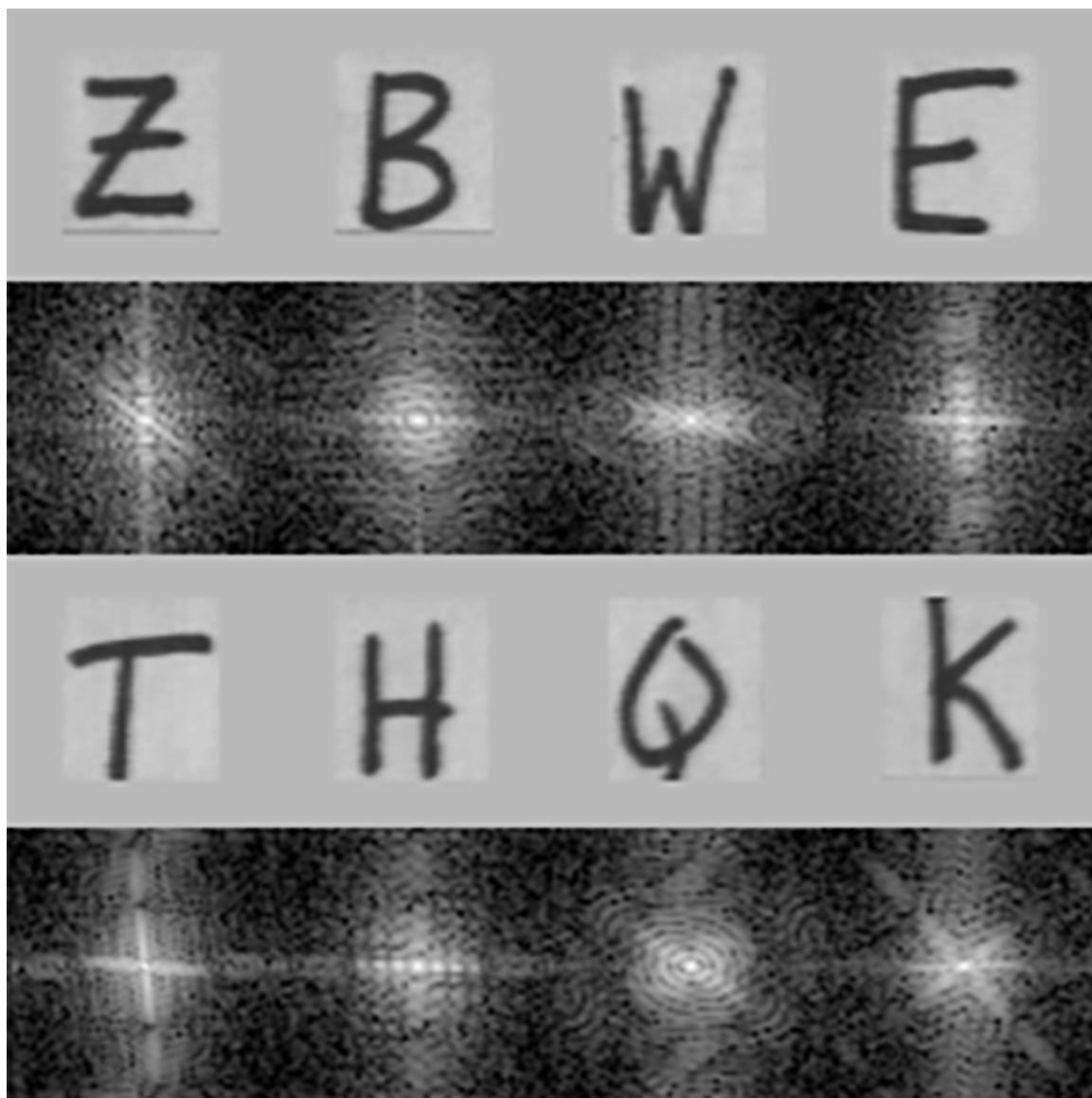
- Automatic image quality grading
- Helping professional photographers sort through 100s to 1000s of photos during a photoshoot by automatically discarding the blurry or low-quality ones
- Applying OCR to real-time video streams, but only applying the expensive OCR computation to non-blurry frames

**The key takeaway here is that it's always easier to write computer vision code for images captured under ideal conditions.** The wise computer vision practitioner will learn to write code that deals with non-ideal conditions with accuracy and precision.

Instead of handling edge cases where image quality is extremely poor, simply detect and discard the low-quality images (e.g., ones with significant blur).

Such a blur detection procedure could either automatically discard the low-quality images or simply tell the end-user “Hey bud, try again. Let’s capture a better image here.”

Keep in mind that computer vision applications are meant to be intelligent, hence the term *artificial intelligence* — and sometimes, that “intelligence” can be detected when input data is of low quality or not rather than trying to make sense of it. Figure 13.1 shows letters directly above their Fourier transform.



**Figure 13.1.** How can we use OpenCV and the fast Fourier transform (FFT) algorithm to detect whether a photo is blurry automatically? (Image credit: John M. Brayer, <http://pyimg.co/3yu9e> [49].)

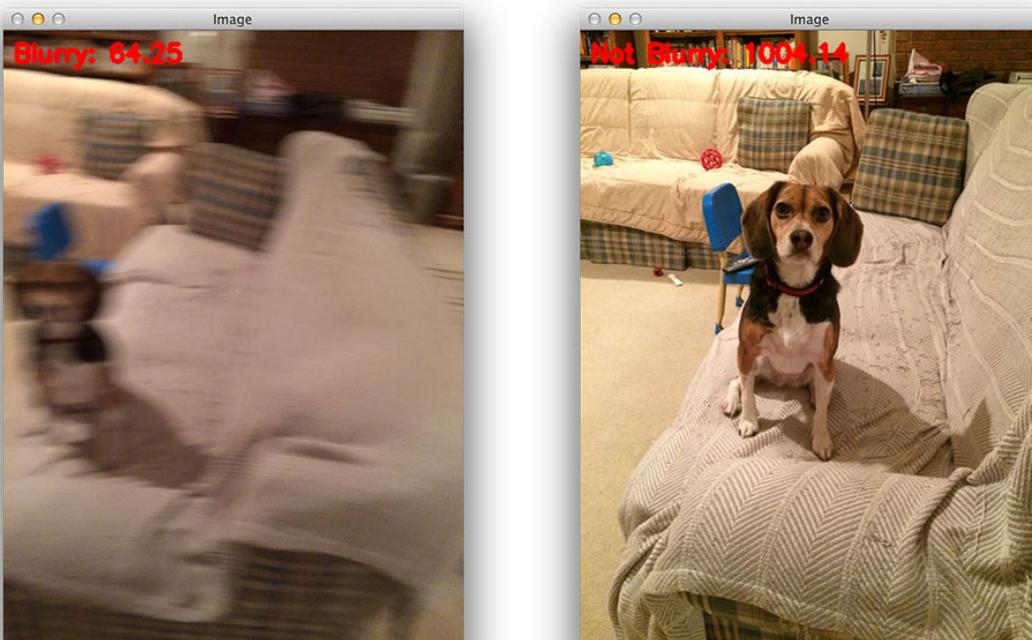
### 13.2.2 What Is the Fast Fourier Transform?

The FFT is a convenient mathematical algorithm for computing the discrete Fourier transform (DFT). It is used for converting a signal from one domain into another. You can see how it is used to detect blur in Figure 13.2.

The FFT is useful in many disciplines, ranging from music, mathematics, science, and engineering. For example, electrical engineers, particularly those working with wireless, power, and audio signals, need the FFT calculation to convert time-series signals into the frequency domain. Some calculations are more easily made in the frequency domain. Conversely, a frequency domain signal could be converted back into the time domain using the FFT.

In terms of computer vision, we often think of the FFT as an image processing tool that represents an image in two domains:

- i. Fourier (i.e., frequency) domain
- ii. Spatial domain



**Figure 13.2.** We'll use a combination of OpenCV and NumPy to conduct FFT-based blur detection in images and video streams in this chapter.

Therefore, the FFT represents the image in both real and imaginary components.

We can perform image processing routines such as blurring, edge detection, thresholding, texture analysis, and *yes, even blur detection by analyzing these values.*

Reviewing the mathematical details of the FFT is outside the scope of this chapter. Hence, if you're interested in learning more about it, I suggest you read the following article from the University of Edinburgh for more details: <http://pyimg.co/2q5rs> [50].

For readers who are academically inclined, take a look at Aaron Bobick's fantastic slides from Georgia Tech's computer vision course (<http://pyimg.co/qvuzm>) [51].

The curious reader may also watch the following video: <http://pyimg.co/fft23> [52].

Finally, the Wikipedia page on the Fourier transform goes into more detail on the mathematics including its applications to non-image processing tasks (<http://pyimg.co/qlshs> [53]).

## 13.3 Implementing Our Text Blur Detector

In this section, we will implement our FFT-based blur detector. We'll start by reviewing our directory structure for the project. We'll then implement a helper function, `detect_blur_fft`, which, as the name suggests, will allow us to perform blur detection in images.

We'll use this function to implement a second Python script, `detect_blur.py`, which will load an input image and then apply the blur detection procedure.

In the next chapter, you will learn how to use `detect_blur_fft` to detect blur in video streams, discard the blurry ones, and only OCR the high-quality frames, thereby increasing your OCR accuracy.

### 13.3.1 Project Structure

Let's start by reviewing our directory structure for the project:

---

```
|-- images
|   |-- example_01.png
|   |-- example_02.png
|   |-- example_03.png
|-- pyimagesearch
|   |-- __init__.py
|   |-- blur_detection
|       |-- __init__.py
|       |-- blur_detector.py
|-- detect_blur.py
```

---

Our FFT-based blur detector algorithm is housed inside the `blur_detection` submodule of `pyimagesearch`. Inside, a single function, `detect_blur_fft` is implemented.

We'll then use the `detect_blur_fft` function inside our driver script, `detect_blur.py`.

The `images` directory contains several sample images to which we can apply text blur detection.

### 13.3.2 Creating the Blur Detector Helper Function

The blur detection method we'll be covering is based on GitHub user, whdcumt's implementation (<http://pyimg.co/omy9p> [54]) of Liu et al.'s 2008 CVPR paper, *Image Partial Blur Detection and Classification* [55].

Open the `blur_detector.py` file in our directory structure, and insert the following code:

---

```

1 # import the necessary packages
2 import numpy as np
3
4 def detect_blur_fft(image, size=60, thresh=10):
5     # grab the dimensions of the image and use the dimensions to
6     # derive the center (x, y)-coordinates
7     (h, w) = image.shape
8     (cX, cY) = (int(w / 2.0), int(h / 2.0))

```

---

Our blur detector implementation requires that we use the NumPy library. We'll use an FFT algorithm built-in to NumPy as the basis for our methodology; we accompany the FFT calculation with additional math as well.

**Line 4** defines the `detect_blur_fft` function, accepting three parameters:

- `image`: Our input image for blur detection
- `size`: The size of the radius around the centerpoint of the image for which we will zero out the FFT shift
- `thresh`: A value which the mean value of the magnitudes (more on that later) will be compared to for determining whether an image is considered blurry or not blurry

Given our input `image`, first we grab its dimensions (**Line 7**) and compute the center  $(x, y)$ -coordinates (**Line 8**).

Next, we'll calculate the DFT using NumPy's implementation of the FFT algorithm:

---

```

10     # compute the FFT to find the frequency transform, then shift
11     # the zero-frequency component (i.e., DC component located at

```

---

---

```

12      # the top-left corner) to the center, where it will be more
13      # easy to analyze
14      fft = np.fft.fft2(image)
15      fftShift = np.fft.fftshift(fft)

```

---

Here, using NumPy's built-in algorithm, we compute the FFT (**Line 14**). We then shift the zero frequency component of the result to the center for easier analysis (**Line 15**).

Next, let's zero out the center of the FFT shift:

---

```

17      # zero out the center of the FFT shift (i.e., remove low
18      # frequencies), apply the inverse shift such that the DC
19      # component once again becomes the top-left, and then apply
20      # the inverse FFT
21      fftShift[cY - size:cY + size, cX - size:cX + size] = 0
22      fftShift = np.fft.ifftshift(fftShift)
23      recon = np.fft.ifft2(fftShift)

```

---

Here, we:

- Zero out the center of our FFT shift (i.e., to remove low frequencies) via **Line 21**
- Apply the inverse shift to put the DC component back in the *top-left* (**Line 22**)
- Apply the inverse FFT (**Line 23**)

And from here, we have three more steps to determine if our `image` is blurry:

---

```

25      # compute the magnitude spectrum of the reconstructed image,
26      # then compute the mean of the magnitude values
27      magnitude = 20 * np.log(np.abs(recon))
28      mean = np.mean(magnitude)
29
30      # the image will be considered "blurry" if the mean value of the
31      # magnitudes is less than the threshold value
32      return (mean, mean <= thresh)

```

---

The remaining steps include:

- Computing the magnitude spectrum, once again, of the reconstructed image after we have already zeroed out the center DC values (**Line 27**).
- Calculating the mean of the magnitude representation (**Line 28**).

- Returning a 2-tuple of the `mean` value and a Boolean indicating whether the input image is blurry or not (**Line 32**). Looking at the code, we can observe that we've determined the blurry Boolean (whether or not the `image` is blurry) by comparing the `mean` to the `thresh` (threshold).

Great job implementing an FFT-based blurriness detector algorithm; we aren't done yet, though. In the next section, we'll apply our algorithm to static images to ensure it is performing to our expectations.

### 13.3.3 Detecting Text and Document Blur

Now that our `detect_blur_fft` helper function is implemented let's put it to use by creating a Python driver script that loads an input image from disk and then applies FFT blur detection to it.

Open a new file, name it `detect_blur_image.py`, and insert the following code:

---

```

1 # import the necessary packages
2 from pyimagesearch.blur_detection import detect_blur_fft
3 import numpy as np
4 import argparse
5 import imutils
6 import cv2

```

---

**Lines 2–6** begin with handling our imports; in particular, we need our `detect_blur_fft` function that we implemented in the previous section.

Next, let's parse our command line arguments:

---

```

8 # construct the argument parser and parse the arguments
9 ap = argparse.ArgumentParser()
10 ap.add_argument("-i", "--image", type=str, required=True,
11     help="path input image that we'll detect blur in")
12 ap.add_argument("-t", "--thresh", type=int, default=15,
13     help="threshold for our blur detector to fire")
14 args = vars(ap.parse_args())

```

---

We only need two command line arguments here:

- i. `--image`: The path to the input image for blur detection.
- ii. `--thresh`: The threshold for our blur detector calculation.

We default the `--thresh` to 15, a good initial value used to test for blur detection that should work in several applications (note that you *may* have to fine-tune this threshold value for your applications).

If you haven't already guessed, the `--image` and `--thresh` arguments correspond to the `image` and `thresh`, and parameters of our `detect_blur_fft` function implemented in the previous section.

Let's go ahead and load our input `--image` and perform the FFT blur detection:

---

```

16 # load the input image from disk, resize it, and convert it to
17 # grayscale
18 orig = cv2.imread(args["image"])
19 orig = imutils.resize(orig, width=500)
20 gray = cv2.cvtColor(orig, cv2.COLOR_BGR2GRAY)
21
22 # apply our blur detector using the FFT
23 (mean, blurry) = detect_blur_fft(gray, size=60,
24     thresh=args["thresh"])

```

---

To conduct the FFT blur detection, we load the input `--image` from disk and convert it to grayscale (**Lines 18–20**). We then apply our FFT blur detector using our `detect_blur_fft` function (**Lines 23 and 24**).

Next, we'll annotate and display our image:

---

```

26 # draw on the image, indicating whether or not it is blurry
27 image = np.dstack([gray] * 3)
28 color = (0, 0, 255) if blurry else (0, 255, 0)
29 text = "Blurry {:.4f}" if blurry else "Not Blurry {:.4f}"
30 text = text.format(mean)
31 cv2.putText(image, text, (10, 25), cv2.FONT_HERSHEY_SIMPLEX,
32     0.7, color, 2)
33 print("[INFO] {}".format(text))
34
35 # show the output image
36 cv2.imshow("Output", image)
37 cv2.waitKey(0)

```

---

In the above code block, we:

- Add two more channels to our single-channel `gray` image, storing the result in `image` (**Line 27**)
- Set the `color` as *red* (if blurry) and *green* (if not blurry) via **Line 28**
- Draw our blurry `text` indication and `mean` value in the *top-left* corner of our `image` (**Lines 30–32**) and `print` the same information in our terminal (**Line 33**)

- Show the output `image` until a key is pressed (**Lines 36 and 37**)

And that's all there is to it! Congrats on implementing your very own blur detector.

### 13.3.4 Blurry Text and Document Detection Results

Let's put our FFT blur detector to work! Open a shell and execute the following command:

---

```
$ python detect_blur.py --image images/example_01.png
[INFO] Blurry (11.2666)
```

---

Here you can see that our script has marked `example_01.png` as “blurry,” and if you check Figure 13.3 (top-left), you can verify that the text on the concert stub ticket is indeed quite blurry — it would be near impossible for any OCR engine, Tesseract or otherwise, to correctly OCR this image.



**Figure 13.3.** *Top-left:* An example of a concert stub that is too blurry for us to OCR. *Top-right:* Another example of blurry text in the image — this example is far more blurry than the one in the *top-left*. *Bottom:* A good quality image without blur that we'll be able to reliably OCR in the following chapter.

Let's try another image of the same concert stub:

---

```
$ python detect_blur.py --image images/example_02.png
[INFO] Blurry (7.1089)
```

---

Take a look at Figure 13.3 (*top-right*), and you can confirm that this image is also blurry, and in fact, is *more* blurry than the first one. Empirically, our FFT returns a smaller mean value — the smaller the value, the *more* blurry the image is. This relationship is reflected in the output.

Let's try one final image, this one of an image that is *not* blurry:

---

```
$ python detect_blur.py --image images/example_03.png
[INFO] Not Blurry (23.0490)
```

---

Figure 13.3 (*bottom*) shows the result of our script. Here, our image has a larger mean FFT value, indicating that our concert stub is *not* blurry. We could then safely take this image and pass it into our OCR engine, confident that we're more likely to obtain a correct OCR result.

## 13.4 Summary

In this chapter, you learned how to detect blurry documents and text using the fast Fourier transform (FFT). While not as simple as our variance of the Laplacian blur detector (<http://pyimg.co/iI68y> [48]), the FFT blur detector is more robust and tends to provide better blur detection accuracy in real-life applications.

I suggest using this method when building computer vision applications that require users to input their photos manually. An example of such an application could be a mobile bank check deposit app that requires a user to snap photos of a check's front and back. Using the FFT, you could determine if the image is low-quality or contains blur and then request that the reader take a new photo of the check.

In our next chapter, you'll learn how to use the FFT method to detect blurry frames in a video stream, thereby allowing us to OCR only the high-quality frames.



## Chapter 14

# OCR'ing Video Streams

In our previous chapter, you learned how to use the fast Fourier transform (FFT) to detect blur in images and documents. Using this method, we were able to detect blurry, low-quality images and then alert the user that they should attempt to capture a higher-quality version so that we can OCR it.

**Remember, it's always easier to write computer vision code that operates on *high-quality* images rather than *low-quality* images.** Using the FFT blur detection method helps ensure that only higher-quality images enter our pipeline.

However, there is another use of the FFT blur detector — it can be used to discard low-quality frames from video streams that would be otherwise impossible to OCR.

Video streams naturally have low-quality frames due to rapid changes in lighting conditions (e.g., walking on a bright sunny day into a dark room), the camera lens autofocusing, or most commonly, *motion blur*.

It would be near impossible to OCR these frames, so instead of attempting to OCR every frame in a video stream (which would lead to nonsensical results for the low-quality frames), but if we could instead simply *detect* that the frame was blurry, ignore it, and then only OCR the high-quality frames?

Is such an implementation possible?

You bet it is — and we'll be covering how to apply our blur detector to OCR video streams in the remainder of this chapter.

### 14.1 Chapter Learning Objectives

Inside this chapter, you will:

- Learn how to OCR video streams
- Apply our FFT blur detector to detect and discard blurry, low-quality frames
- Build an output visualization script that shows the stages of OCR'ing video streams
- Put all the pieces together and fully implement OCR in video streams

## 14.2 OCR'ing Real-Time Video Streams

In the first part of this chapter, we will review our project directory structure.

We'll then implement a simple video writer utility class. This class will allow us to create an *output* video of our blur detection and OCR results from an *input* video.

Given our video writer helper function, we'll then implement our driver script to apply OCR to video streams.

We'll wrap up this chapter with a discussion of our results.

### 14.2.1 Project Structure

Let's get started by reviewing the directory structure for our video OCR project:

---

```
|-- pyimagesearch
|   |-- __init__.py
|   |-- helpers.py
|   |-- blur_detection
|       |-- __init__.py
|       |-- blur_detector.py
|   |-- video_ocr
|       |-- __init__.py
|       |-- visualization.py
|-- output
|   |-- ocr_video_output.avi
|-- video
|   |-- business_card.mp4
|-- ocr_video.py
```

---

Inside the `pyimagesearch` module, we have two submodules:

- i. `blur_detection`: The submodule which we'll be using to aid in detecting blur in video streams.
- ii. `video_ocr`: Contains a helper function that writes the output of our video OCR to disk as a separate video file.

The `video` directory contains `business_card.mp4`, a video containing a business card that we want to OCR. The `output` directory contains the output of running the driver script, `ocr_video.py`, on our input video.

### 14.2.2 Implementing Our Video Writer Utility

Before we implement our driver script, we first need to implement a basic helper utility that will allow us to write the output of our video OCR script to disk as a *separate* output video.

A sample of the output of the visualization script can be seen in Figure 14.1. Notice that the output has three components:

- i. The original input frame with the business card detected and blurry/not blurry annotation (*top*)
- ii. The *top-down* transform of the business card with the text detected (*middle*)
- iii. The OCR'd text itself from the *top-down* transform (*bottom*)

We'll be implementing a helper utility function to build such an output visualization in this section.

Please note that this video writer utility has *nothing* to do with OCR. Instead, it's just a simple Python class I implemented to write in video I/O. I'm merely reviewing it in this chapter as a matter of completeness.

If you find yourself struggling to follow along with the class implementation, *don't worry* — it will not impact your OCR knowledge. That said, if you would like to learn more about working with video and OpenCV, I recommend you follow my “*Working with Video*” tutorials on the PyImageSearch blog: <http://pyimg.co/s6wiy> [56].

Let's get started implementing our video writer utility now. Open the `visualization.py` file in the `video_ocr` directory of our project, and let's get started:

---

```
 1 # import the necessary packages
 2 import numpy as np
 3
 4 class VideoOCROutputBuilder:
 5     def __init__(self, frame):
 6         # store the input frame dimensions
 7         self.maxW = frame.shape[1]
 8         self.maxH = frame.shape[0]
```

---



**Figure 14.1.** Our video writer utility will output a video file with three rows, including the raw video frame with blurry/not blurry detection (*top*), the perspective transform of the detected business card with text bounding boxes (*middle*), and the OCR'd text itself (*bottom*). We use this video writer utility so that we can better visualize our video OCR results.

We start by defining our `VideoOCROutputBuilder` class. Our constructor requires only a single parameter, our input `frame`. We then store the width and height of the `frame` as `maxW` and `maxH`, respectively.

With our constructor taken care of, let's create the `build` method responsible for constructing the visualization you saw in Figure 14.1.

---

```

10  def build(self, frame, card=None, ocr=None):
11      # grab the input frame dimensions and initialize the card
12      # image dimensions along with the OCR image dimensions
13      (frameH, frameW) = frame.shape[:2]
14      (cardW, cardH) = (0, 0)
15      (ocrW, ocrH) = (0, 0)
16
17      # if the card image is not empty, grab its dimensions
18      if card is not None:
19          (cardH, cardW) = card.shape[:2]
20
21      # similarly, if the OCR image is not empty, grab its
22      # dimensions
23      if ocr is not None:
24          (ocrH, ocrW) = ocr.shape[:2]

```

---

The `build` method accepts three arguments, one of which is required (the other two are optional):

- i. `frame`: The input frame from the video
- ii. `card`: The business card after a *top-down* perspective transform has been applied, and the text on the card detected
- iii. `ocr`: The OCR'd text itself

**Line 13** grabs the spatial dimensions of the input `frame` while **Lines 14 and 15** initialize the `card` and `ocr` images' spatial dimensions, respectively.

Since both `card` and `ocr` could be `None`, we don't know if they are valid images. If they *are*, **Lines 18–24** makes this check, and if it passes, grabs the width and height of the `card` and `ocr`.

We can now start constructing our output visualization:

---

```

26      # compute the spatial dimensions of the output frame
27      outputW = max([frameW, cardW, ocrW])
28      outputH = frameH + cardH + ocrH
29
30      # update the max output spatial dimensions found thus far
31      self.maxW = max(self.maxW, outputW)
32      self.maxH = max(self.maxH, outputH)
33
34      # allocate memory of the output image using our maximum

```

---

---

```

35     # spatial dimensions
36     output = np.zeros((self.maxH, self.maxW, 3), dtype="uint8")
37
38     # set the frame in the output image
39     output[0:frameH, 0:frameW] = frame

```

---

**Line 27** computes the maximum *width* of the `output` visualization by finding the `max` height across the `frame`, `card`, and `ocr`. **Line 28** determines the *height* of the visualization by adding all three heights together (we do this addition operation because these images need to be *stacked*, on top of the other).

**Lines 31 and 32** update our `maxW` and `maxH` bookkeeping variables with the largest width and height values we've found thus far.

Given our newly updated `maxW` and `maxH`, **Line 36** allocates memory for our `output` image using the maximum spatial dimensions we found thus far.

With the `output` image initialized, we store the `frame` at the top of the `output` (**Line 39**).

Our next code block handles adding the `card` and the `ocr` images to the `output` frame:

---

```

41     # if the card is not empty, add it to the output image
42     if card is not None:
43         output[frameH:frameH + cardH, 0:cardW] = card
44
45     # if the OCR result is not empty, add it to the output image
46     if ocr is not None:
47         output[
48             frameH + cardH:frameH + cardH + ocrH,
49             0:ocrW] = ocr
50
51     # return the output visualization image
52     return output

```

---

**Lines 42 and 43** verify that a valid `card` image has been passed into the function, and if so, we add it to the `output` image. **Lines 45–49** do the same, only for the `ocr` image.

Finally, we return the `output` visualization to the calling function.

Congrats on implementing our `VideoOCROutputBuilder` class! Let's put it to work in the next section!

### 14.2.3 Implementing Our Real-Time Video OCR Script

We are now ready to implement our `ocr_video.py` script. Let's get to work:

---

```

1 # import the necessary packages
2 from pyimagesearch.video_ocr import VideoOCROutputBuilder
3 from pyimagesearch.blur_detection import detect_blur_fft
4 from pyimagesearch.helpers import cleanup_text
5 from imutils.video import VideoStream
6 from imutils.perspective import four_point_transform
7 from pytesseract import Output
8 import pytesseract
9 import numpy as np
10 import argparse
11 import imutils
12 import time
13 import cv2

```

---

We start on **Lines 2–13** importing our required Python packages. The notable imports include:

- `VideoOCROutputBuilder`: Our visualization builder
- `detect_blur_fft`: Our FFT blur detector
- `cleanup_text`: Used to clean up OCR'd text, stripping out non-ASCII characters, such that we can draw the OCR'd text on the output image using OpenCV's `cv2.putText` function
- `four_point_transform`: Applies a perspective transform such that we can obtain a *top-down/bird's-eye view* of the business card we're OCR'ing
- `pytesseract`: Provides an interface to the Tesseract OCR engine

With our imports taken care of, let's move on to our command line arguments:

---

```

15 # construct the argument parser and parse the arguments
16 ap = argparse.ArgumentParser()
17 ap.add_argument("-i", "--input", type=str,
18     help="path to optional input video (webcam will be used otherwise)")
19 ap.add_argument("-o", "--output", type=str,
20     help="path to optional output video")
21 ap.add_argument("-c", "--min-conf", type=int, default=50,
22     help="minimum confidence value to filter weak text detection")
23 args = vars(ap.parse_args())

```

---

Our script provides three command line arguments:

- i. `--input`: Path to an optional input video file on disk. If no video file is provided, we'll use our webcam.
- ii. `--output`: Path to an optional output video file that we'll generate.

iii. `--min-conf`: Minimum confidence value used to filter out weak text detections.

Now we can move on to our initializations:

---

```

25 # initialize our video OCR output builder used to easily visualize
26 # output to our screen
27 outputBuilder = None
28
29 # initialize our output video writer along with the dimensions of the
30 # output frame
31 writer = None
32 outputW = None
33 outputH = None

```

---

**Line 27** initializes our `outputBuilder`. This object will be instantiated in the main body of our `while` loop that accesses frames from our video stream (which we'll cover later in this chapter).

We then initialize the output video writer and the spatial dimensions of the output video on **Lines 31–33**.

Let's move on to accessing our video stream:

---

```

35 # create a named window for our output OCR visualization (a named
36 # window is required here so that we can automatically position it
37 # on our screen)
38 cv2.namedWindow("Output")
39
40 # initialize a Boolean used to indicate if either a webcam or input
41 # video is being used
42 webcam = not args.get("input", False)
43
44 # if a video path was not supplied, grab a reference to the webcam
45 if webcam:
46     print("[INFO] starting video stream...")
47     vs = VideoStream(src=0).start()
48     time.sleep(2.0)
49
50 # otherwise, grab a reference to the video file
51 else:
52     print("[INFO] opening video file...")
53     vs = cv2.VideoCapture(args["input"])

```

---

**Line 38** creates a named window called `Output` for our output visualization. We *explicitly* created a named window here so we can use OpenCV's `cv2.moveWindow` function to move the window on our screen. We need to perform this moving operation because the size of the output window is *dynamic* — it will grow in size as our output grows and shrinks.

**Line 42** determines if we are using a webcam as video input or not. If so, **Lines 45–48** access our webcam video stream; otherwise, **Lines 51–53** grab a pointer to the video residing on disk.

With access to our video stream, it's now time to start looping over frames:

---

```

55 # loop over frames from the video stream
56 while True:
57     # grab the next frame and handle if we are reading from either
58     # a webcam or a video file
59     orig = vs.read()
60     orig = orig if webcam else orig[1]
61
62     # if we are viewing a video and we did not grab a frame then we
63     # have reached the end of the video
64     if not webcam and orig is None:
65         break
66
67     # resize the frame and compute the ratio of the *new* width to
68     # the *old* width
69     frame = imutils.resize(orig, width=600)
70     ratio = orig.shape[1] / float(frame.shape[1])
71
72     # if our video OCR output builder is None, initialize it
73     if outputBuilder is None:
74         outputBuilder = VideoOCROutputBuilder(frame)

```

---

**Lines 59 and 60** read the original (`orig`) frame from the video stream. If the `webcam` variable is set and the `orig` frame is `None`, we have reached the end of the video file, so we break from the loop.

Otherwise, **Lines 69 and 70** resize the frame to have a width of 700 pixels (such that it's easier and faster to process) and then compute the `ratio` of the *new* width to the *old* width — we'll need this ratio when applying the perspective transform to the *original high-resolution* frame later in this loop.

**Lines 73 and 74** initialize our `VideoOCROutputBuilder` using the resized frame.

Next comes a few more initializations, followed by blur detection:

---

```

76     # initialize our card and OCR output ROIs
77     card = None
78     ocr = None
79
80     # convert the frame to grayscale and detect if the frame is
81     # considered blurry or not
82     gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
83     (mean, blurry) = detect_blur_fft(gray, thresh=15)

```

---

---

```

84      # draw whether or not the frame is blurry
85      color = (0, 0, 255) if blurry else (0, 255, 0)
86      text = "Blurry {:.4f}" if blurry else "Not Blurry {:.4f}"
87      text = text.format(mean)
88      cv2.putText(frame, text, (10, 25), cv2.FONT_HERSHEY_SIMPLEX,
89                  0.7, color, 2)

```

---

**Lines 77 and 78** initialize our `card` and `ocr` ROIs. The `card` ROI will contain the *top-down* transform of the business card (if the business card is found in the current `frame`), while `ocr` will contain the OCR'd text itself.

We then perform text/document blur detection on **Lines 82 and 83**. We first convert the `frame` to grayscale and then apply our `detect_blur_fft` function.

**Lines 86–90** draw on the `frame`, indicating whether or not the current frame is blurry.

Let's continue with our video OCR pipeline:

---

```

92      # only continue to process the frame for OCR if the image is
93      # *not* blurry
94      if not blurry:
95          # blur the grayscale image slightly and then perform edge
96          # detection
97          blurred = cv2.GaussianBlur(gray, (5, 5), 0)
98          edged = cv2.Canny(blurred, 75, 200)
99
100         # find contours in the edge map and sort them by size in
101         # descending order, keeping only the largest ones
102         cnts = cv2.findContours(edged.copy(), cv2.RETR_EXTERNAL,
103                             cv2.CHAIN_APPROX_SIMPLE)
104         cnts = imutils.grab_contours(cnts)
105         cnts = sorted(cnts, key=cv2.contourArea, reverse=True)[:5]
106
107         # initialize a contour that corresponds to the business card
108         # outline
109         cardCnt = None
110
111         # loop over the contours
112         for c in cnts:
113             # approximate the contour
114             peri = cv2.arcLength(c, True)
115             approx = cv2.approxPolyDP(c, 0.02 * peri, True)
116
117             # if our approximated contour has four points, then we
118             # can assume we have found the outline of the business
119             # card
120             if len(approx) == 4:
121                 cardCnt = approx
122                 break

```

---

Before continuing, we make a check to verify that the frame is *not* blurry. Provided the check passes, we start finding the business card in the input frame by smoothing the frame with a Gaussian kernel and then applying edge detection (**Lines 97 and 98**).

We then apply contour detection to the edge map and sort the contours by their area, from the largest to smallest (**Lines 102–105**). Our assumption here is that the business card will be the largest ROI in the input frame that *also* has four vertices.

To determine if we've found the business card or not, we loop over the largest contours on **Line 112**. We then apply contour approximation (**Lines 114 and 115**) and then check to see if the approximated contour has four points.

Provided the contour has four points, we assume we've found our card contour, so we store the contour variable (`cardCnt`) and then `break` from the loop (**Lines 120–122**).

If we found our business card contour, we now attempt to OCR it:

---

```

124         # ensure that the business card contour was found
125     if cardCnt is not None:
126         # draw the outline of the business card on the frame so
127         # we visually verify that the card was detected correctly
128         cv2.drawContours(frame, [cardCnt], -1, (0, 255, 0), 3)
129
130         # apply a four-point perspective transform to the
131         # *original* frame to obtain a top-down bird's-eye
132         # view of the business card
133         card = four_point_transform(orig,
134             cardCnt.reshape(4, 2) * ratio)
135
136         # allocate memory for our output OCR visualization
137         ocr = np.zeros(card.shape, dtype="uint8")
138
139         # swap channel ordering for the business card and OCR it
140         rgb = cv2.cvtColor(card, cv2.COLOR_BGR2RGB)
141         results = pytesseract.image_to_data(rgb,
142             output_type=Output.DICT)

```

---

**Line 125** verifies that we have indeed found our business card contour. We then draw the card contour on our `frame` via OpenCV's `cv2.drawContours` function.

Next, we apply a perspective transform to the *original, high-resolution image* (such that we can better OCR it) by using our `four_point_transform` function (**Lines 133 and 134**). We also allocate memory for our output `ocr` visualization, using the same spatial dimensions of the `card` after applying the *top-down* transform (**Line 137**).

**Lines 140–142** then apply text detection and OCR to the business card.

The next step is to annotate the output `ocr` visualization with the OCR'd text itself:

---

```

144         # loop over each of the individual text localizations
145         for i in range(0, len(results["text"])):
146             # extract the bounding box coordinates of the text
147             # region from the current result
148             x = results["left"][i]
149             y = results["top"][i]
150             w = results["width"][i]
151             h = results["height"][i]
152
153             # extract the OCR text itself along with the
154             # confidence of the text localization
155             text = results["text"][i]
156             conf = int(results["conf"][i])
157
158             # filter out weak confidence text localizations
159             if conf > args["min_conf"]:
160                 # process the text by stripping out non-ASCII
161                 # characters
162                 text = cleanup_text(text)
163
164                 # if the cleaned up text is not empty, draw a
165                 # bounding box around the text along with the
166                 # text itself
167                 if len(text) > 0:
168                     cv2.rectangle(card, (x, y), (x + w, y + h),
169                               (0, 255, 0), 2)
170                     cv2.putText(ocr, text, (x, y - 10),
171                               cv2.FONT_HERSHEY_SIMPLEX, 0.5,
172                               (0, 0, 255), 1)

```

---

**Line 145** loops over all text detections. We then proceed to:

- Grab the bounding box coordinates of the text ROI (**Lines 148–151**)
- Extract the OCR'd text and its corresponding confidence/probability (**Lines 155 and 156**)
- Verify that the text detection has sufficient confidence, followed by stripping out non-ASCII characters from the text (**Lines 159–162**)
- Draw the OCR'd text on the `ocr` visualization (**Lines 167–172**)

The rest of the code blocks in this example focus more on bookkeeping variables and output:

---

```

174     # build our final video OCR output visualization
175     output = outputBuilder.build(frame, card, ocr)
176
177     # check if the video writer is None *and* an output video file
178     # path was supplied
179     if args["output"] is not None and writer is None:

```

---

```

180      # grab the output frame dimensions and initialize our video
181      # writer
182      (outputH, outputW) = output.shape[:2]
183      fourcc = cv2.VideoWriter_fourcc(*"MJPG")
184      writer = cv2.VideoWriter(args["output"], fourcc, 27,
185                               (outputW, outputH), True)
186
187      # if the writer is not None, we need to write the output video
188      # OCR visualization to disk
189      if writer is not None:
190          # force resize the video OCR visualization to match the
191          # dimensions of the output video
192          outputFrame = cv2.resize(output, (outputW, outputH))
193          writer.write(outputFrame)
194
195      # show the output video OCR visualization
196      cv2.imshow("Output", output)
197      cv2.moveWindow("Output", 0, 0)
198      key = cv2.waitKey(1) & 0xFF
199
200      # if the `q` key was pressed, break from the loop
201      if key == ord("q"):
202          break

```

---

**Line 175** creates our `output` frame using the `.build` method our `VideoOCROutputBuilder` class.

We then check to see if an `--output` video file path is supplied, and if so, instantiate our `cv2.VideoWriter` so we can write the `output` frame visualizations to disk (**Lines 179–185**).

Similarly, if the `writer` object has been instantiated, we then write the `output` frame to disk (**Lines 189–193**).

**Lines 196–202** display the `output` frame to our screen:

Our final code block releases video pointers:

```

204      # if we are using a webcam, stop the camera video stream
205      if webcam:
206          vs.stop()
207
208      # otherwise, release the video file pointer
209      else:
210          vs.release()
211
212      # close any open windows
213      cv2.destroyAllWindows()

```

---

Taken as a whole, this may seem like a complicated script. But keep in mind that we just

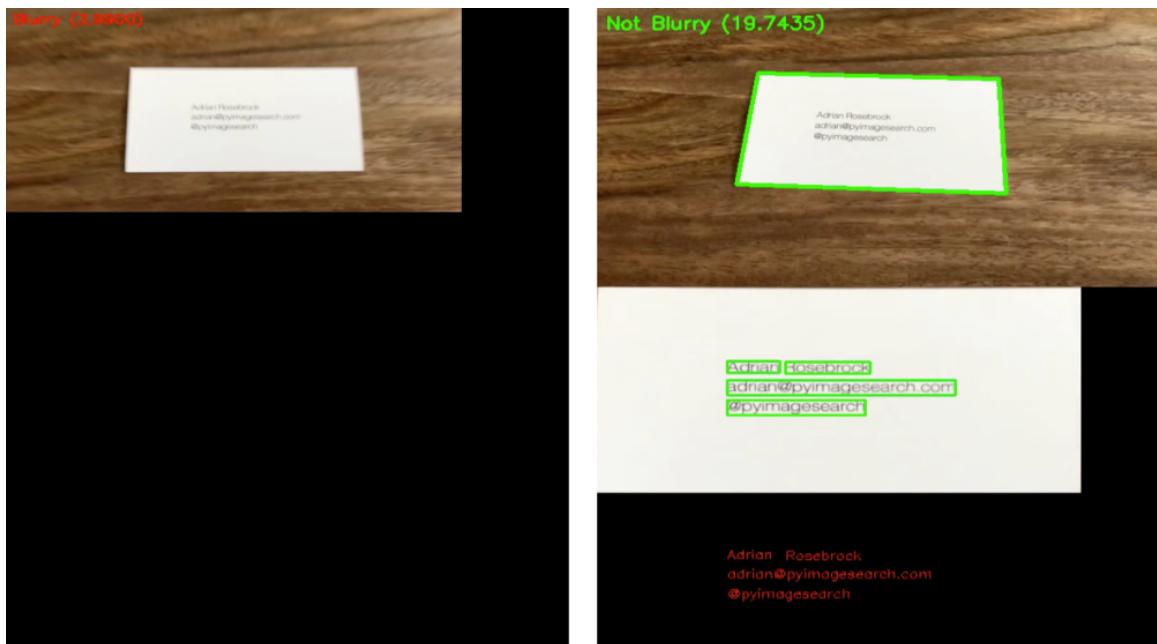
implemented an *entire* video OCR pipeline in under 225 lines of code (including comments). That's not much code when you think about it — and it's all made possible by using OpenCV and Tesseract!

#### 14.2.4 Real-Time Video OCR Results

We are now ready to put our video OCR script to the test! Open a terminal and execute the following command:

```
$ python ocr_video.py --input video/business_card.mp4 --output
→   output/ocr_video_output.avi
[INFO] opening video file...
```

I clearly cannot include a video demo or animated graphics interchange format (GIF) inside a book/PDF, so I've taken screen captures from our `ocr_video_output.avi` file in the `output` directory — the screen captures can be seen in Figure 14.2.



**Figure 14.2.** *Left:* Detecting a frame that is *too blurry* to OCR. Instead of attempting to OCR this frame, which would lead to incorrect or nonsensical results, we instead wait for a higher-quality frame. *Right:* Detecting a frame that is of *sufficient quality* to OCR. Here you see that we have correctly OCR'd the business card.

Notice on the *left* that our script has correctly detected a blurry frame and did not OCR it. If we had attempted to OCR this frame, the results would have been nonsensical, confusing the end-user.

Instead, we wait for a higher-quality frame (*right*) and then OCR it. As you can see, by waiting for the higher-quality frame, we were able to correctly OCR the business card.

If you ever need to apply OCR to video streams, I *strongly recommend* that you implement some type of low-quality versus high-quality frame detector. **Do not attempt to OCR every single frame of the video stream unless you are 100% confident that the video was captured under ideal, controlled conditions and that every frame is high-quality.**

### 14.3 Summary

In this chapter, you learned how to OCR video streams. However, to OCR the video streams, we need to detect blurry, low-quality frames.

Videos naturally have low-quality frames due to rapid changes in lighting conditions, camera lens autofocusing, and motion blur. Instead of trying to OCR these low-quality frames, which would ultimately lead to low OCR accuracy (or worse, totally nonsensical results), we instead need to *detect* these low-quality frames and discard them.

One easy way to detect low-quality frames is to use blur detection. We utilized our FFT blur detector to work with our video stream. The result is an OCR pipeline capable of operating on video streams while still maintaining high accuracy.

I hope you enjoyed this chapter! And I hope that you can apply this method to your projects.



## Chapter 15

# Improving Text Detection Speed with OpenCV and GPUs

up to this point, everything except the EasyOCR material has focused on performing OCR on our CPU — **but what if we could instead apply OCR on our GPU?** Since many state-of-the-art text detection and OCR models are deep learning-based, couldn't these models run faster and more efficiently on a GPU?

The answer is *yes*; they absolutely can.

This chapter will show you how to take the efficient and accurate scene text detector (EAST) model and run it on OpenCV's `dnn` (deep neural network) module using an NVIDIA GPU. As we'll see, our text detection throughput rate nearly triples, improving from  $\approx 23$  frames per second (FPS) to an astounding  $\approx 97$  FPS!

### 15.1 Chapter Learning Objectives

In this chapter, you will:

- Learn how to use OpenCV's `dnn` module to run deep neural networks on an NVIDIA CUDA-based GPU
- Implement a Python script to benchmark text detection speed on both a CPU and GPU
- Implement a second Python script, this one that performs text detection in real-time video streams
- Compare the results of running text detection on a CPU versus a GPU

## 15.2 Using Your GPU for OCR with OpenCV

The first part of this chapter covers reviewing our directory structure for the project.

We'll then implement a Python script that will benchmark running text detection on a CPU versus a GPU. We'll run this script and measure just how much of a difference running text detection on a GPU improves our FPS throughput rate.

Once we've measured our FPS increase, we'll implement a second Python script, this one to perform text detection in real-time video streams.

We'll wrap up the chapter with a discussion of our results.

### 15.2.1 Project Structure

Before we can apply text detection with our GPU, we first need to review our project directory structure:

---

```
|-- pyimagesearch
|   |-- __init__.py
|   |-- east
|       |-- __init__.py
|       |-- east.py
|-- ../models
|   |-- east
|       |-- frozen_east_text_detection.pb
-- images
|   |-- car_wash.png
|-- text_detection_speed.py
|-- text_detection_video.py
```

---

We'll be reviewing two Python scripts in this chapter:

- i. `text_detection_speed.py`: Benchmarks text detection speed on a CPU versus a GPU using the `car_wash.png` image in our `images` directory.
- ii. `text_detection_video.py`: Demonstrates how to perform real-time text detection on your GPU.

### 15.2.2 Implementing Our OCR GPU Benchmark Script

Before we implement text detection in real-time video streams with our GPU, let's first *benchmark* how much of a speedup we get by running the EAST detection model on our CPU versus our GPU.

To find out, open the `text_detection_speed.py` file in our project directory, and let's get started:

---

```
1 # import the necessary packages
2 from pyimagesearch.east import EAST_OUTPUT_LAYERS
3 import numpy as np
4 import argparse
5 import time
6 import cv2
```

---

**Lines 2–6** handle importing our required Python packages. We need the EAST model's output layers (**Line 2**) to grab the text detection outputs. If you need a refresher on these output values, be sure to refer to “*OCR with OpenCV, Tesseract, and Python: Intro to OCR*” book.

Next, we have our command line arguments:

---

```
8 # construct the argument parser and parse the arguments
9 ap = argparse.ArgumentParser()
10 ap.add_argument("-i", "--image", required=True,
11                  help="path to input image")
12 ap.add_argument("-e", "--east", required=True,
13                  help="path to input EAST text detector")
14 ap.add_argument("-w", "--width", type=int, default=320,
15                  help="resized image width (should be multiple of 32)")
16 ap.add_argument("-t", "--height", type=int, default=320,
17                  help="resized image height (should be multiple of 32)")
18 ap.add_argument("-c", "--min-conf", type=float, default=0.5,
19                  help="minimum probability required to inspect a text region")
20 ap.add_argument("-n", "--nms-thresh", type=float, default=0.4,
21                  help="non-maximum suppression threshold")
22 ap.add_argument("-g", "--use-gpu", type=bool, default=False,
23                  help="boolean indicating if CUDA GPU should be used")
24 args = vars(ap.parse_args())
```

---

The `--image` command line argument specifies the path to the input image that we'll be performing text detection on.

**Lines 12–21** then specify command line arguments for the EAST text detection model.

Finally, we have our `--use-gpu` command line argument. By default, we'll use our CPU — but by specifying this argument (and provided we have a CUDA-capable GPU and OpenCV's `dnn` module compiled with NVIDIA GPU support), then we can use our GPU for text detection inference.

With our command line arguments taken care of, we can now load the EAST text detection model and set whether we are using the CPU or GPU:

---

```

26 # load the pre-trained EAST text detector
27 print("[INFO] loading EAST text detector...")
28 net = cv2.dnn.readNet(args["east"])
29
30 # check if we are going to use GPU
31 if args["use_gpu"]:
32     # set CUDA as the preferable backend and target
33     print("[INFO] setting preferable backend and target to CUDA...")
34     net.setPreferableBackend(cv2.dnn.DNN_BACKEND_CUDA)
35     net.setPreferableTarget(cv2.dnn.DNN_TARGET_CUDA)
36
37 # otherwise we are using our CPU
38 else:
39     print("[INFO] using CPU for inference...")

```

---

**Line 28** loads our EAST text detection model from disk.

**Lines 31–35** makes a check to see if the `--use-gpu` command line argument was supplied, and if so, indicates that we want to use our NVIDIA CUDA-capable GPU.

**Note:** To use your GPU for neural network inference, you need to have OpenCV's `dnn` module compiled with NVIDIA CUDA support. OpenCV's `dnn` module does not have NVIDIA support via a pip install. Instead, you need to compile OpenCV with GPU support explicitly. I cover how to do that in the tutorial on PyImageSearch: [pyimg.co/jftm](http://pyimg.co/jftm) [57].

Next, let's load our sample image from disk:

---

```

41 # load the input image and then set the new width and height values
42 # based on our command line arguments
43 image = cv2.imread(args["image"])
44 (newW, newH) = (args["width"], args["height"])
45
46 # construct a blob from the image, set the blob as input to the
47 # network, and initialize a list that records the amount of time
48 # each forward pass takes
49 print("[INFO] running timing trials...")
50 blob = cv2.dnn.blobFromImage(image, 1.0, (newW, newH),
51     (123.68, 116.78, 103.94), swapRB=True, crop=False)
52 net.setInput(blob)
53 timings = []

```

---

**Line 43** loads our input `--image` from disk while **Lines 50 and 51** construct a `blob` object such that we can pass it through the EAST text detection model.

**Line 52** sets our `blob` as input to the EAST network, while **Line 53** initializes a `timings` list to measure how long the inference takes.

When using a GPU for inference, your first prediction tends to be very slow compared to the

rest of the predictions, the reason being that your GPU hasn't "warmed up" yet. Therefore, when taking measurements on your GPU, you typically want to take an average over several predictions.

In the following code block, we perform text detection for 500 trials, recording how long each prediction takes:

---

```
55 # loop over 500 trials to obtain a good approximation to how long
56 # each forward pass will take
57 for i in range(0, 500):
58     # time the forward pass
59     start = time.time()
60     (scores, geometry) = net.forward(EAST_OUTPUT_LAYERS)
61     end = time.time()
62     timings.append(end - start)
63
64 # show average timing information on text prediction
65 avg = np.mean(timings)
66 print("[INFO] avg. text detection took {:.6f} seconds".format(avg))
```

---

After all trials are complete, we compute the average of the `timings` and then display our average text detection time on our terminal.

### 15.2.3 Speed Test: OCR With and Without GPU

Let's now measure our EAST text detection FPS throughput rate *without* a GPU (i.e., running on a CPU):

---

```
$ python text_detection_speed.py --image images/car_wash.png --east
↪ ./models/east/frozen_east_text_detection.pb
[INFO] loading EAST text detector...
[INFO] using CPU for inference...
[INFO] running timing trials...
[INFO] avg. text detection took 0.108568 seconds
```

---

Our average text detection speed is  $\approx 0.1$  seconds, which equates to  $\approx 9\text{--}10$  FPS. A deep learning model running on a CPU is quite fast and sufficient for many applications.

However, as Tim Taylor (played by Tim Allen of *Toy Story*) from the 1990s TV show, *Home Improvement*, says, "*More power!*"

Let's now break out the GPUs:

---

```
$ python text_detection_speed.py --image images/car_wash.png --east
↪ ./models/east/frozen_east_text_detection.pb --use-gpu 1
```

---

---

```
[INFO] loading EAST text detector...
[INFO] setting preferable backend and target to CUDA...
[INFO] running timing trials...
[INFO] avg. text detection took 0.004763 seconds
```

---

Using an NVIDIA V100 GPU, our average frame processing rate decreases to  $\approx 0.004$  seconds, meaning that we can now process  $\approx 250$  FPS! **As you can see, using your GPU makes a *substantial* difference!**

#### 15.2.4 OCR on GPU for Real-Time Video Streams

Ready to implement our script to perform text detection in real-time video streams using your GPU?

Open the `text_detection_video.py` file in your project directory, and let's get started:

---

```
1 # import the necessary packages
2 from pyimagesearch.east import EAST_OUTPUT_LAYERS
3 from pyimagesearch.east import decode_predictions
4 from imutils.video import VideoStream
5 from imutils.video import FPS
6 import numpy as np
7 import argparse
8 import imutils
9 import time
10 import cv2
```

---

**Lines 2–10** import our required Python packages. The `EAST_OUTPUT_LAYERS` and `decode_predictions` function come from our implementation of the EAST text detector in Chapter 18, “*Rotated Text Bounding Box Localization with OpenCV*” from Book One of this two-part book series — be sure to review that chapter if you need a refresher on the EAST detection model.

**Line 4** imports our `VideoStream` to access our webcam, while **Line 5** provides our `FPS` class to measure the FPS throughput rate of our pipeline.

Let's now proceed to our command line arguments:

---

```
12 # construct the argument parser and parse the arguments
13 ap = argparse.ArgumentParser()
14 ap.add_argument("-i", "--input", type=str,
15                 help="path to optional input video file")
16 ap.add_argument("-e", "--east", required=True,
17                 help="path to input EAST text detector")
```

---

---

```

18 ap.add_argument("-w", "--width", type=int, default=320,
19     help="resized image width (should be multiple of 32)")
20 ap.add_argument("-t", "--height", type=int, default=320,
21     help="resized image height (should be multiple of 32)")
22 ap.add_argument("-c", "--min-conf", type=float, default=0.5,
23     help="minimum probability required to inspect a text region")
24 ap.add_argument("-n", "--nms-thresh", type=float, default=0.4,
25     help="non-maximum suppression threshold")
26 ap.add_argument("-g", "--use-gpu", type=bool, default=False,
27     help="boolean indicating if CUDA GPU should be used")
28 args = vars(ap.parse_args())

```

---

These command line arguments are nearly the same as previous command line arguments, the only exception is that we swapped out the `--image` command line argument for an `--input` argument, which specifies the path to an optional video file on disk (just in case we wanted to use a video file rather than our webcam).

Next, we have a few initializations:

---

```

30 # initialize the original frame dimensions, new frame dimensions,
31 # and ratio between the dimensions
32 (W, H) = (None, None)
33 (newW, newH) = (args["width"], args["height"])
34 (rW, rH) = (None, None)

```

---

Here we initialize our original frame's width and height, the new frame dimensions for the EAST model, followed by the ratio between the *original* and the *new* dimensions.

This next code block handles loading the EAST text detection model from disk and then setting whether or not we are using our CPU or GPU for inference:

---

```

36 # load the pre-trained EAST text detector
37 print("[INFO] loading EAST text detector...")
38 net = cv2.dnn.readNet(args["east"])

39 # check if we are going to use GPU
40 if args["use_gpu"]:
41     # set CUDA as the preferable backend and target
42     print("[INFO] setting preferable backend and target to CUDA...")
43     net.setPreferableBackend(cv2.dnn.DNN_BACKEND_CUDA)
44     net.setPreferableTarget(cv2.dnn.DNN_TARGET_CUDA)

45 # otherwise we are using our CPU
46 else:
47     print("[INFO] using CPU for inference...")

```

---

Our text detection model needs frames to operate on, so the next code block accesses either

our webcam or a video file residing on disk, depending on whether or not the `--input` command line argument was supplied:

---

```

51 # if a video path was not supplied, grab the reference to the webcam
52 if not args.get("input", False):
53     print("[INFO] starting video stream...")
54     vs = VideoStream(src=0).start()
55     time.sleep(1.0)
56
57 # otherwise, grab a reference to the video file
58 else:
59     vs = cv2.VideoCapture(args["input"])
60
61 # start the FPS throughput estimator
62 fps = FPS().start()

```

---

**Line 62** starts measuring our FPS throughput rates to get a good idea of the number of frames our text detection pipeline can process in a single second.

Let's start looping over frames from the video stream now:

---

```

64 # loop over frames from the video stream
65 while True:
66     # grab the current frame, then handle if we are using a
67     # VideoStream or VideoCapture object
68     frame = vs.read()
69     frame = frame[1] if args.get("input", False) else frame
70
71     # check to see if we have reached the end of the stream
72     if frame is None:
73         break
74
75     # resize the frame, maintaining the aspect ratio
76     frame = imutils.resize(frame, width=1000)
77     orig = frame.copy()
78
79     # if our frame dimensions are None, we still need to compute the
80     # ratio of old frame dimensions to new frame dimensions
81     if W is None or H is None:
82         (H, W) = frame.shape[:2]
83         rW = W / float(newW)
84         rH = H / float(newH)

```

---

**Lines 68 and 69** read the next frame from either our webcam or our video file.

If we are indeed processing a video file, **Lines 72** makes a check to see if we are at the end of the video and if so, we break from the loop.

**Lines 81–84** grab the spatial dimensions of the input `frame` and then compute the ratio of the original frame dimensions to the dimensions required by the EAST model.

Now that we have these dimensions, we can construct our input to the EAST text detector:

---

```

86      # construct a blob from the image and then perform a forward pass
87      # of the model to obtain the two output layer sets
88      blob = cv2.dnn.blobFromImage(frame, 1.0, (newW, newH),
89          (123.68, 116.78, 103.94), swapRB=True, crop=False)
90      net.setInput(blob)
91      (scores, geometry) = net.forward(EAST_OUTPUT_LAYERS)
92
93      # decode the predictions form OpenCV's EAST text detector and
94      # then apply non-maximum suppression (NMS) to the rotated
95      # bounding boxes
96      (rects, confidences) = decode_predictions(scores, geometry,
97          minConf=args["min_conf"])
98      idxs = cv2.dnn.NMSBoxesRotated(rects, confidences,
99          args["min_conf"], args["nms_thresh"])

```

---

**Lines 88–91** build `blob` from the input `frame`. We then set this `blob` as input to our EAST text detection `net`. A forward pass of the network is performed, resulting in our raw text detections.

However, our raw text detections are unusable in our current state, so we call `decode_predictions` on them, yielding a 2-tuple of the bounding box coordinates of the text detections along with the associated probabilities (**Lines 96 and 97**).

We then apply non-maxima suppression to suppress weak, overlapping bounding boxes (otherwise, there would be *multiple* bounding boxes for each detection).

If you need more details on this code block, including how the `decode_predictions` function is implemented, be sure to review Chapter 18, *Rotated Text Bounding Box Localization with OpenCV* of the “*Intro to OCR*” Bundle, where I cover the EAST text detector in far more detail.

After non-maximum suppression (NMS), we can now loop over each of the bounding boxes:

---

```

101     # ensure that at least one text bounding box was found
102     if len(idxs) > 0:
103         # loop over the valid bounding box indexes after applying NMS
104         for i in idxs.flatten():
105             # compute the four corners of the bounding box, scale the
106             # coordinates based on the respective ratios, and then
107             # convert the box to an integer NumPy array
108             box = cv2.boxPoints(rects[i])
109             box[:, 0] *= rW

```

---

```

110         box[:, 1] *= rH
111         box = np.int0(box)
112
113         # draw a rotated bounding box around the text
114         cv2.polyline(orig, [box], True, (0, 255, 0), 2)
115
116         # update the FPS counter
117         fps.update()
118
119         # show the output frame
120         cv2.imshow("Text Detection", orig)
121         key = cv2.waitKey(1) & 0xFF
122
123         # if the `q` key was pressed, break from the loop
124         if key == ord("q"):
125             break

```

---

**Line 102** verifies that at least one text bounding box was found, and if so, we loop over the indexes of the kept bounding boxes after applying NMS.

For each resulting index, we compute the bounding box of the text ROI, scale the bounding box  $(x, y)$ -coordinates back to the `orig` input frame dimensions, and then draw the bounding box on the `orig` frame (**Lines 108–114**).

**Line 117** updates our FPS throughput estimator while **Lines 120–125** display the output text detection on our screen.

The final step here is to stop our FPS time, approximate the throughput rate, and release any video file pointers:

```

127     # stop the timer and display FPS information
128     fps.stop()
129     print("[INFO] elapsed time: {:.2f}".format(fps.elapsed()))
130     print("[INFO] approx. FPS: {:.2f}".format(fps.fps()))
131
132     # if we are using a webcam, release the pointer
133     if not args.get("input", False):
134         vs.stop()
135
136     # otherwise, release the file pointer
137     else:
138         vs.release()
139
140     # close all windows
141     cv2.destroyAllWindows()

```

---

**Lines 128–130** stop our FPS timer and approximate the FPS of our text detection pipeline. We then release any video file pointers and close any windows opened by OpenCV.

### 15.2.5 GPU and OCR Results

This section needs to be executed locally on a machine with a GPU. After running the `text_detection_video.py` script on an NVIDIA RTX 2070 SUPER GPU (coupled with an i9 9900K processor), I obtained  $\approx 97$  FPS:

---

```
$ python text_detection_video.py --east
↪ ./models/east/frozen_east_text_detection.pb --use-gpu 1
[INFO] loading EAST text detector...
[INFO] setting preferable backend and target to CUDA...
[INFO] starting video stream...
[INFO] elapsed time: 74.71
[INFO] approx. FPS: 96.80
```

---

When I ran the same script *without* using any GPU, I reached an FPS of  $\approx 23$ , which is  $\approx 77\%$  *slower* than the above results.

---

```
$ python text_detection_video.py --east
↪ ./models/east/frozen_east_text_detection.pb
[INFO] loading EAST text detector...
[INFO] using CPU for inference...
[INFO] starting video stream...
[INFO] elapsed time: 68.59
[INFO] approx. FPS: 22.70
```

---

As you can see, using your GPU can *dramatically* improve the throughput speed of your text detection pipeline!

## 15.3 Summary

In this chapter, you learned how to perform text detection in real-time video streams using your GPU. Since many text detection and OCR models are deep learning-based, using your GPU (versus your CPU) can tremendously increase your frame processing throughput rate.

Using our CPU, we were able to process  $\approx 22$ – $23$  FPS. However, by running the EAST model on OpenCV's `dnn` module, we could reach  $\approx 97$  FPS!

If you have a GPU available to you, definitely consider utilizing it — you'll be able to run text detection models in real-time!



## Chapter 16

# Text Detection and OCR with Amazon Rekognition API

So far, we've primarily focused on using the Tesseract OCR engine. However, other optical character recognition (OCR) engines are available, some of which are *far more accurate* than Tesseract and capable of accurately OCR'ing text, even in complex, unconstrained conditions.

Typically, these OCR engines live in the cloud. Many are proprietary-based. To keep these models and associated datasets proprietary, the companies do not distribute the models themselves and instead put them behind a REST API.

While these models do tend to be more accurate than Tesseract, there are some downsides, including:

- An internet connection is required to OCR images — that's less of an issue for most laptops/desktops, but if you're working on the edge, an internet connection may not be possible
- Additionally, if you are working with edge devices, then you may not want to spend the power draw on a network connection
- There will be latency introduced by the network connection
- OCR results will take longer because the image has to be packaged into an API request and uploaded to the OCR API. The API will need to chew on the image and OCR it, and then finally return the results to the client
- Due to the latency and amount of time it will take to OCR each image, it's doubtful that these OCR APIs will be able to run in real-time
- They cost money (but typically offer a free trial or are free up to a number of monthly API requests)

Looking at the previous list, you may be wondering why on earth would we cover these APIs at all — what is the benefit?

As you'll see, the primary benefit here is *accuracy*. Consider the amount of data that Google and Microsoft have from running their respective search engines. Then consider the amount of data Amazon generates *daily* from simply printing shipping labels.

**These companies have an *incredible* amount of image data — and when they train their novel, state-of-the-art OCR models on their data, the result is an *incredibly* robust and accurate OCR model.**

In this chapter, you'll learn how to use the Amazon Rekognition API to OCR images. We'll then cover Microsoft Azure Cognitive Services in Chapter 17 and Google Cloud Vision API in Chapter 18.

## 16.1 Chapter Learning Objectives

In this chapter, you will:

- Learn about the Amazon Rekognition API
- Discover how the Amazon Rekognition API can be used for OCR
- Obtain your Amazon Web Services (AWS) Rekognition Keys
- Install Amazon's `boto3` package to interface with the OCR API
- Implement a Python script that interfaces with Amazon Rekognition API to OCR an image

## 16.2 Amazon Rekognition API for OCR

The first part of this chapter will focus on obtaining your AWS Rekognition Keys. These keys will include a public access key and a secret key, similar to SSH, SFTP, etc.

I'll then show you how to install the `boto3`, the Amazon Web Services (AWS) software development kit (SDK) for Python. We'll use the `boto3` package to interface with Amazon Rekognition OCR API.

Next, we'll implement our Python configuration file (which will store our access key, secret key, and AWS region) and then create our driver script used to:

- i. Load an input image from disk

- ii. Package it into an API request
- iii. Send the API request to AWS Rekognition for OCR
- iv. Retrieve the results from the API call
- v. Display our OCR results

We'll wrap up this chapter with a discussion of our results.

### 16.2.1 Obtaining Your AWS Rekognition Keys

You will need additional information from our companion site for instructions on how to obtain your AWS Rekognition keys. You can find the instructions here: <http://pyimg.co/vxd51>.

### 16.2.2 Installing Amazon's Python Package

To interface with the Amazon Rekognition API, we need to use the `boto3` package: the AWS SDK. Luckily, `boto3` is incredibly simple to install, requiring only a single `pip-install` command:

---

```
$ pip install boto3
```

---

If you are using a Python virtual environment or an Anaconda environment, be sure to use the appropriate command to access your Python environment *before* running the above command (otherwise, `boto3` will be installed in the system install of Python).

### 16.2.3 Project Structure

Before we can perform text detection and OCR with Amazon Rekognition API, we first need to review our project directory structure.

---

```
|-- config
|   |-- __init__.py
|   |-- aws_config.py
|-- images
|   |-- aircraft.png
|   |-- challenging.png
|   |-- park.png
|   |-- street_signs.png
|-- amazon_ocr.py
```

---

The `aws_config.py` file contains our AWS access key, secret key, and region. You learned how to obtain these values in Section 16.2.1.

Our `amazon_ocr.py` script will take this `aws_config`, connect to the Amazon Rekognition API, and then perform text detection and OCR to each image in our `images` directory.

#### 16.2.4 Creating Our Configuration File

To connect to the Amazon Rekognition API, we first need to supply our access key, secret key, and region. If you haven't yet obtained these keys, go to Section 16.2.1 and be sure to follow the steps and note the values.

Afterward, you can come back here, open `aws_config.py`, and update the code:

---

```
1 # define our AWS Access Key, Secret Key, and Region
2 ACCESS_KEY = "YOUR_ACCESS_KEY"
3 SECRET_KEY = "YOUR_SECRET_KEY"
4 REGION = "YOUR_AWS_REGION"
```

---

Sharing my API keys would be a security breach, so I've left placeholder values here. **Be sure to update them with your API keys; otherwise, you will be unable to connect to the Amazon Rekognition API.**

#### 16.2.5 Implementing the Amazon Rekognition OCR Script

With our `aws_config` implemented, let's move on to the `amazon_ocr.py` script, which is responsible for:

- i. Connecting to the Amazon Rekognition API
- ii. Loading an input image from disk
- iii. Packaging the image in an API request
- iv. Sending the package to Amazon Rekognition API for OCR
- v. Obtaining the OCR results from Amazon Rekognition API
- vi. Displaying our output text detection and OCR results

Let's get started with our implementation:

---

```
1 # import the necessary packages
2 from config import aws_config as config
3 import argparse
4 import boto3
5 import cv2
```

---

**Lines 2–5** import our required Python packages. Notably, we need our `aws_config` (defined in the previous section), along with `boto3`, which is Amazon's Python package to interface with their API.

Let's now define `draw_ocr_results`, a simple Python utility used to draw the output OCR results from Amazon Rekognition API:

---

```
7 def draw_ocr_results(image, text, poly, color=(0, 255, 0)):
8     # unpack the bounding box, taking care to scale the coordinates
9     # relative to the input image size
10    (h, w) = image.shape[:2]
11    t1X = int(poly[0]["X"] * w)
12    t1Y = int(poly[0]["Y"] * h)
13    trX = int(poly[1]["X"] * w)
14    trY = int(poly[1]["Y"] * h)
15    brX = int(poly[2]["X"] * w)
16    brY = int(poly[2]["Y"] * h)
17    blX = int(poly[3]["X"] * w)
18    blY = int(poly[3]["Y"] * h)
```

---

The `draw_ocr_results` function accepts four parameters:

- i. `image`: The input image that we're drawing the OCR'd text on
- ii. `text`: The OCR'd text itself
- iii. `poly`: The polygon object/coordinates of the text bounding box returned by Amazon Rekognition API
- iv. `color`: The color of the bounding box

**Line 10** grabs the width and height of the `image` that we're drawing on. **Lines 11–18** then grab the bounding box coordinates of the text ROI, taking care to scale the coordinates by the width and height.

Why do we perform this scaling process?

Well, as we'll find out later in this chapter, Amazon Rekognition API returns bounding boxes in the range  $[0, 1]$ . Multiplying the bounding boxes by the original image width and height brings the bounding boxes back to the original image scale.

From there, we can now annotate the `image`:

---

```

20  # build a list of points and use it to construct each vertex
21  # of the bounding box
22  pts = ((tlX, tlY), (trX, trY), (brX, brY), (blX, blY))
23  topLeft = pts[0]
24  topRight = pts[1]
25  bottomRight = pts[2]
26  bottomLeft = pts[3]
27
28  # draw the bounding box of the detected text
29  cv2.line(image, topLeft, topRight, color, 2)
30  cv2.line(image, topRight, bottomRight, color, 2)
31  cv2.line(image, bottomRight, bottomLeft, color, 2)
32  cv2.line(image, bottomLeft, topLeft, color, 2)
33
34  # draw the text itself
35  cv2.putText(image, text, (topLeft[0], topLeft[1] - 10),
36               cv2.FONT_HERSHEY_SIMPLEX, 0.8, color, 2)
37
38  # return the output image
39  return image

```

---

**Lines 22–26** builds a list of points which correspond to each vertex of the bounding box. Given the vertices, **Lines 29–32** draw the bounding box of the rotated text. **Lines 35 and 36** draw the OCR'd text itself.

We then return the annotated output `image` to the calling function.

With our helper utility defined, let's move on to command line arguments:

---

```

41  # construct the argument parser and parse the arguments
42  ap = argparse.ArgumentParser()
43  ap.add_argument("-i", "--image", required=True,
44                  help="path to input image that we'll submit to AWS Rekognition")
45  ap.add_argument("-t", "--type", type=str, default="line",
46                  choices=["line", "word"],
47                  help="output text type (either 'line' or 'word')")
48  args = vars(ap.parse_args())

```

---

The `--image` command line argument corresponds to the path to the input image we want to submit to the Amazon Rekognition OCR API.

The `--type` argument can be either `line` or `word`, indicating whether or not we want the Amazon Rekognition API to return OCR'd results grouped into *lines* or as individual *words*.

Next, let's connect to Amazon Web Services:

---

```

50 # connect to AWS so we can use the Amazon Rekognition OCR API
51 client = boto3.client(
52     "rekognition",
53     aws_access_key_id=config.ACCESS_KEY,
54     aws_secret_access_key=config.SECRET_KEY,
55     region_name=config.REGION)
56
57 # load the input image as a raw binary file and make a request to
58 # the Amazon Rekognition OCR API
59 print("[INFO] making request to AWS Rekognition API...")
60 image = open(args["image"], "rb").read()
61 response = client.detect_text(Image={"Bytes": image})
62
63 # grab the text detection results from the API and load the input
64 # image again, this time in OpenCV format
65 detections = response["TextDetections"]
66 image = cv2.imread(args["image"])
67
68 # make a copy of the input image for final output
69 final = image.copy()

```

---

**Lines 51–55** connect to AWS. Here we supply our access key, secret key, and region.

Once connected, we load our input image from disk as a binary object (**Line 60**) and then submit it to AWS by calling the `detect_text` function and supplying our `image`.

Calling `detect_text` results in a `response` from the Amazon Rekognition API. We then grab the `TextDetections` results (**Line 65**).

**Line 66** loads the input `--image` from disk in OpenCV format, while **Line 69** clones the image to draw on it.

We can now loop over the text detection bounding boxes from the Amazon Rekognition API:

---

```

71 # loop over the text detection bounding boxes
72 for detection in detections:
73     # extract the OCR'd text, text type, and bounding box coordinates
74     text = detection["DetectedText"]
75     textType = detection["Type"]
76     poly = detection["Geometry"]["Polygon"]
77
78     # only draw show the output of the OCR process if we are looking
79     # at the correct text type
80     if args["type"] == textType.lower():
81         # draw the output OCR line-by-line
82         output = image.copy()
83         output = draw_ocr_results(output, text, poly)
84         final = draw_ocr_results(final, text, poly)
85
86     # show the output OCR'd line

```

---

---

```

87     print(text)
88     cv2.imshow("Output", output)
89     cv2.waitKey(0)
90
91 # show the final output image
92 cv2.imshow("Final Output", final)
93 cv2.waitKey(0)

```

---

**Line 72** loops over all detections returned by Amazon's OCR API. We then extract the OCR'd text, `textType` (either “`word`” or “`line`”), along with the bounding box coordinates of the OCR'd text (**Lines 74 and 75**).

**Line 80** makes a check to verify whether we are looking at either `word` or `line` OCR'd text. If the current `textType` matches our `--type` command line argument, we call our `draw_ocr_results` function on both the `output` image and our `final` cloned image (**Lines 82–84**).

**Lines 87–89** display the current OCR'd line or word on our terminal and screen. That way, we can easily visualize each line or word without the output image becoming too cluttered.

Finally, **Lines 92 and 93** show the result of drawing *all* text on our screen at once (for visualization purposes).

### 16.2.6 Amazon Rekognition OCR Results

Congrats on implementing a Python script to interface with Amazon Rekognition's OCR API!

Let's see our results in action, first by OCR'ing the entire image, line-by-line:

---

```
$ python amazon_ocr.py --image images/aircraft.png
[INFO] making request to AWS Rekognition API...
WARNING!
LOW FLYING AND DEPARTING AIRCRAFT
BLAST CAN CAUSE PHYSICAL INJURY
```

---

Figure 16.1 shows that we have OCR'd our input `aircraft.png` image line-by-line successfully, thereby demonstrating that the Amazon Rekognition API was able to:

- i. Locate each block of text in the input image
- ii. OCR each text ROI
- iii. Group the blocks of text into lines

But what if we wanted to obtain our OCR results at the *word* level instead of the *line* level?

That's as simple as supplying the `--type` command line argument:

```
$ python amazon_ocr.py --image images/aircraft.png --type word  
[INFO] making request to AWS Rekognition API...
```

```
WARNING!  
LOW  
FLYING  
AND  
DEPARTING  
AIRCRAFT  
BLAST  
CAN  
CAUSE  
PHYSICAL  
INJURY
```

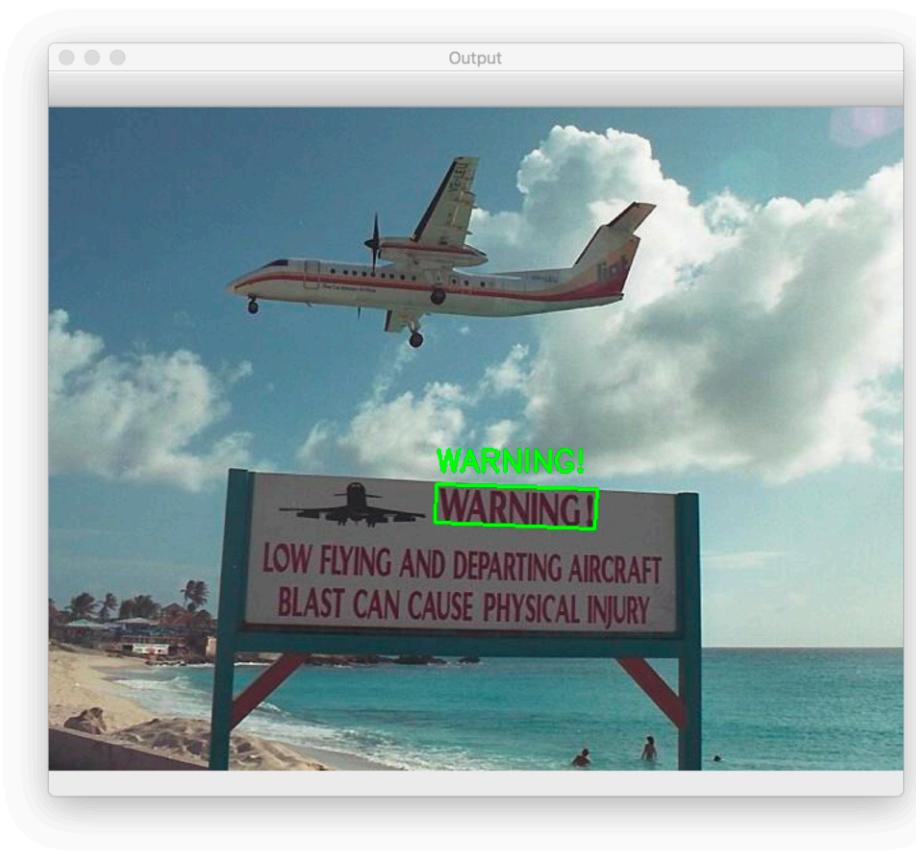


**Figure 16.1.** OCR'ing images line-by-line using the Amazon Rekognition API.

As our output and Figure 16.2 shows, we're now OCR'ing text at the *word* level.

I'm a big fan of Amazon Rekognition's OCR API. While AWS, EC2, etc., do have a bit of a learning curve, the benefit is that once you know it, you then understand Amazon's entire web services ecosystem, which makes it *far easier* to start integrating different services.

I also appreciate that it's super easy to get started with Amazon Rekognition API. Other services (e.g., Google Cloud Vision API) make it a bit harder to get that "first easy win." If this is your first foray into using cloud service APIs, definitely consider using Amazon Rekognition API first before moving on to Microsoft Cognitive Services or the Google Cloud Vision API.



**Figure 16.2.** OCR'ing an image word-by-word using the Amazon Rekognition OCR API.

### 16.3 Summary

In this chapter, you learned how to create your Amazon Rekognition keys, install `boto3`, a Python package used to interface with AWS, and then implemented a Python script to make calls to the Amazon Rekognition API.

The Python script was simple and straightforward, requiring less than 100 lines to implement (including comments).

Not only were our Amazon Rekognition OCR API results correct, but we could also parse the results at both the *line* and *word* level, giving us finer granularity than what the EAST text detection model and Tesseract OCR engine give us (at least without fine-tuning several options).

In Chapter 17, we'll look at the Microsoft Cognitive Services API for OCR.



## Chapter 17

# Text Detection and OCR with Microsoft Cognitive Services

In our previous chapter, you learned how to use the Amazon Rekognition API to OCR images. The hardest part of using the Amazon Rekognition API was obtaining your API keys. Once you had your API keys, it was smooth sailing.

This chapter focuses on a different cloud-based API called Microsoft Cognitive Services (MCS), part of Microsoft Azure. Like Amazon Rekognition API, MCS is also capable of high OCR accuracy — but unfortunately, the implementation is slightly more complex (as is both Microsoft's login and admin dashboard).

I prefer the Amazon Web Services (AWS) Rekognition API over MCS, both for the admin dashboard and the API itself. However, if you are already ingrained into the MCS/Azure ecosystem, you should consider staying there. The MCS API isn't *that* hard to use (it's just not as easy and straightforward as Amazon Rekognition API).

### 17.1 Chapter Learning Objectives

In this chapter, you will:

- Learn how to obtain your MCS API keys
- Create a configuration file to store your subscription key and API endpoint URL
- Implement a Python script to make calls to the MCS OCR API
- Review the results of applying the MCS OCR API to sample images

## 17.2 Microsoft Cognitive Services for OCR

We'll start this chapter with a review of how you can obtain your MCS API keys. **You will need these API keys to make requests to the MCS API to OCR images.**

Once we have our API keys, we'll review our project directory structure and then implement a Python configuration file to store our subscription key and OCR API endpoint URL.

With our configuration file implemented, we'll move on to creating a second Python script, this one acting as a driver script that:

- Imports our configuration file
- Loads an input image to disk
- Packages the image into an API call
- Makes a request to the MCS OCR API
- Retrieves the results
- Annotates our output image
- Displays the OCR results to our screen and terminal

Let's dive in!

### 17.2.1 Obtaining Your Microsoft Cognitive Services Keys

Before proceeding to the rest of the sections, be sure to obtain the API keys by following the instructions shown here: [pyimg.co/53rux](http://pyimg.co/53rux).

### 17.2.2 Project Structure

The directory structure for our MCS OCR API is similar to the structure of the Amazon Rekognition API project in Chapter 16:

---

```
|-- config
|   |-- __init__.py
|   |-- microsoft_cognitive_services.py
|-- images
|   |-- aircraft.png
|   |-- challenging.png
|   |-- park.png
|   |-- street_signs.png
|-- microsoft_ocr.py
```

---

Inside the `config`, we have our `microsoft_cognitive_services.py` file, which stores our subscription key and endpoint URL (i.e., the URL of the API we're submitting our images to).

The `microsoft_ocr.py` script will take our subscription key and endpoint URL, connect to the API, submit the images in our `images` directory for OCR, and then display our screen results.

### 17.2.3 Creating Our Configuration File

Ensure you have followed Section 17.2.1 to obtain your subscription keys to the MCS API. From there, open the `microsoft_cognitive_services.py` file and update your `SUBSCRIPTION_KEY`:

---

```
# define our Microsoft Cognitive Services subscription key
SUBSCRIPTION_KEY = "YOUR_SUBSCRIPTION_KEY"

# define the ACS endpoint
ENDPOINT_URL = "YOUR_ENDPOINT_URL"
```

---

You should replace the string "`YOUR_SUBSCRIPTION_KEY`" with your subscription key obtained from Section 17.2.1.

Additionally, ensure you double-check your `ENDPOINT_URL`. At the time of this writing, the endpoint URL points to the most recent version of the MCS API; however, as Microsoft releases new versions of the API, this endpoint URL may change, so it's worth double-checking.

### 17.2.4 Implementing the Microsoft Cognitive Services OCR Script

Let's now learn how to submit images for text detection and OCR to the MCS API.

Open the `microsoft_ocr.py` script in the project directory structure and insert the following code:

---

```
1 # import the necessary packages
2 from config import microsoft_cognitive_services as config
3 import requests
4 import argparse
5 import time
6 import sys
7 import cv2
```

---

Note on **Line 2** that we import our `microsoft_cognitive_services` configuration to supply our subscription key and endpoint URL. We'll use the `requests` Python package to send requests to the API.

Next, let's define `draw_ocr_results`, a helper function used to annotate our output images with the OCR'd text:

---

```

9  def draw_ocr_results(image, text, pts, color=(0, 255, 0)):
10     # unpack the points list
11     topLeft = pts[0]
12     topRight = pts[1]
13     bottomRight = pts[2]
14     bottomLeft = pts[3]
15
16     # draw the bounding box of the detected text
17     cv2.line(image, topLeft, topRight, color, 2)
18     cv2.line(image, topRight, bottomRight, color, 2)
19     cv2.line(image, bottomRight, bottomLeft, color, 2)
20     cv2.line(image, bottomLeft, topLeft, color, 2)
21
22     # draw the text itself
23     cv2.putText(image, text, (topLeft[0], topLeft[1] - 10),
24                 cv2.FONT_HERSHEY_SIMPLEX, 0.8, color, 2)
25
26     # return the output image
27     return image

```

---

Our `draw_ocr_results` function has four parameters:

- i. `image`: The input image that we are going to draw on.
- ii. `text`: The OCR'd text.
- iii. `pts`: The *top-left*, *top-right*, *bottom-right*, and *bottom-left* (*x*, *y*)-coordinates of the text ROI
- iv. `color`: The BGR color we're using to draw on the `image`

**Lines 11–14** unpack our bounding box coordinates. From there, **Lines 17–20** draw the bounding box surrounding the text in the image. We then draw the OCR'd text itself on **Lines 23–24**.

We wrap up this function by returning the output `image` to the calling function.

We can now parse our command line arguments:

---

```

29  # construct the argument parser and parse the arguments
30  ap = argparse.ArgumentParser()

```

---

---

```

31 ap.add_argument("-i", "--image", required=True,
32     help="path to input image that we'll submit to Microsoft OCR")
33 args = vars(ap.parse_args())
34
35 # load the input image from disk, both in a byte array and OpenCV
36 # format
37 imageData = open(args["image"], "rb").read()
38 image = cv2.imread(args["image"])

```

---

We only need a single argument here, `--image`, which is the path to the input image on disk. We read this image from disk, both as a binary byte array (so we can submit it to the MCS API), and then again in OpenCV/NumPy format (so we can draw on/annotate it).

Let's now construct a request to the MCS API:

---

```

40 # construct our headers dictionary that will include our Microsoft
41 # Cognitive Services API Key (required in order to submit requests
42 # to the API)
43 headers = {
44     "Ocp-Apim-Subscription-Key": config.SUBSCRIPTION_KEY,
45     "Content-Type": "application/octet-stream",
46 }
47
48 # make the request to the Azure Cognitive Services API and wait for
49 # a response
50 print("[INFO] making request to Microsoft Cognitive Services API...")
51 response = requests.post(config.ENDPOINT_URL, headers=headers,
52     data=imageData)
53 response.raise_for_status()
54
55 # initialize whether or not the API request was a success
56 success = False

```

---

**Lines 43–46** define our `headers` dictionary. Note that we are supplying our `SUBSCRIPTION_KEY` here — now is a good time to go back to `microsoft_cognitive_services.py` and ensure you have correctly inserted your subscription key (otherwise, the request to the MCS API will fail).

We then submit the image for OCR to the MCS API on **Lines 51–53**. We also initialize a Boolean, `success`, to indicate if submitting the request was successful or not.

We now have to wait and poll for results from the MCS API:

---

```

58 # continue to poll the Cognitive Services API for a response until
59 # either (1) we receive a "success" response or (2) the request fails
60 while True:
61     # check for a response and convert it to a JSON object

```

```

62     responseFinal = requests.get(
63         response.headers["Operation-Location"],
64         headers=headers)
65     result = responseFinal.json()
66
67     # if the results are available, stop the polling operation
68     if "analyzeResult" in result.keys():
69         success = True
70         break
71
72     # check to see if the request failed
73     if "status" in result.keys() and result["status"] == "failed":
74         break
75
76     # sleep for a bit before we make another request to the API
77     time.sleep(1.0)
78
79 # if the request failed, show an error message and exit
80 if not success:
81     print("[INFO] Microsoft Cognitive Services API request failed")
82     print("[INFO] Attempting to gracefully exit")
83     sys.exit(0)

```

---

I'll be honest — polling for results is not my favorite way to work with an API. It requires more code, it's a bit more tedious, and it can be potentially error-prone if the programmer isn't careful to `break` out of the loop properly.

Of course, there are pros to this approach, including maintaining a connection, submitting larger chunks of data, and having results returned in *batches* rather than *all at once*.

Regardless, this is how Microsoft has implemented its API, so we must play by their rules.

**Line 60** starts a `while` loop that continuously checks for responses from the MCS API (**Lines 62–65**).

If we find the text "`analyzeResult`" in the `result` dictionary's keys, we can safely `break` from the loop and process our results (**Lines 68–70**).

Otherwise, if we find the string "`status`" in the `result` dictionary *and* the status is "`failed`", then the API request failed (and we should `break` from the loop).

If neither of these conditions is met, we `sleep` for a small amount of time and then poll again.

**Lines 80–83** handle the case in which we could not obtain a result from the MCS API. If that happens, then we have no OCR results to show (since the image could not be processed), and then we exit gracefully from our script.

Provided our OCR request succeeded, let's now process the results:

---

```
85 # grab all OCR'd lines returned by Microsoft's OCR API
86 lines = result["analyzeResult"]["readResults"][0]["lines"]
87
88 # make a copy of the input image for final output
89 final = image.copy()
90
91 # loop over the lines
92 for line in lines:
93     # extract the OCR'd line from Microsoft's API and unpack the
94     # bounding box coordinates
95     text = line["text"]
96     box = line["boundingBox"]
97     (tlX, tlY, trX, trY, brX, brY, blX, blY) = box
98     pts = ((tlX, tlY), (trX, trY), (brX, brY), (blX, blY))
99
100    # draw the output OCR line-by-line
101    output = image.copy()
102    output = draw_ocr_results(output, text, pts)
103    final = draw_ocr_results(final, text, pts)
104
105    # show the output OCR'd line
106    print(text)
107    cv2.imshow("Output", output)
108    cv2.waitKey(0)
109
110    # show the final output image
111    cv2.imshow("Final Output", final)
112    cv2.waitKey(0)
```

---

**Line 86** grabs all OCR'd lines returned by the MCS API. **Line 89** initializes our `final` output image with all text drawn on it.

We start looping through all `lines` of OCR'd text on **Line 92**. We extract the OCR'd `text` and bounding box coordinates for each `line`, followed by constructing a list of the *top-left*, *top-right*, *bottom-right*, and *bottom-left* corners, respectively (**Lines 95–98**).

We then draw the OCR'd text line-by-line on the `output` and `final` image (**Lines 101–103**). We display the current line of text on our screen and terminal (**Lines 106–108**) — the final output image, with all OCR'd text drawn on it, is displayed on **Lines 111–112**.

### 17.2.5 Microsoft Cognitive Services OCR Results

Let's now put the MCS OCR API to work for us. Open a terminal and execute the following command:

---

```
$ python microsoft_ocr.py --image images/aircraft.png
[INFO] making request to Microsoft Cognitive Services API...
```

WARNING!  
LOW FLYING AND DEPARTING AIRCRAFT  
BLAST CAN CAUSE PHYSICAL INJURY

Figure 17.1 shows the output of applying the MCS OCR API to our aircraft warning sign. If you recall, this is the same image we used in Chapter 16 when applying the Amazon Rekognition API [58]. I included the same image here in this chapter to demonstrate that the MCS OCR API can correctly OCR this image.



**Figure 17.1.** Image of a plane flying over a warning sign. OCR results displayed on the sign in green.

Let's try a different image, this one containing several challenging pieces of text:

```
$ python microsoft_ocr.py --image images/challenging.png
[INFO] making request to Microsoft Cognitive Services API...
```

LITTER  
EMERGENCY  
First

Eastern National  
Bus Times  
STOP

Figure 17.2 shows the results of applying the MCS OCR API to our input image — and as we can see, MCS does a *great* job OCR’ing the image.



**Figure 17.2.** *Left:* Sample text from the First Eastern National bus timetable. The sample text is a tough image to OCR due to the low image quality and glossy print. Still, Microsoft’s OCR API can correctly OCR it! *Middle:* The Microsoft OCR API can correctly OCR the “Emergency Stop” text. *Right:* A trashcan with the text “Litter.” We’re able to OCR the text, but the text at the bottom of the trashcan is unreadable, even to the human eye.

On the *left*, we have a sample image from the First Eastern National bus timetable (i.e., schedule of when a bus will arrive). The document is printed with a glossy finish (likely to prevent water damage). Still, due to the gloss, there is a significant reflection in the image, particularly in the “*Bus Times*” text. Still, the MCS OCR API can correctly OCR the image.

In the *middle*, the “*Emergency Stop*” text is highly pixelated and low-quality, but that doesn’t phase the MCS OCR API! It’s able to correctly OCR the image.

Finally, the *right* shows a trashcan with the text “*Litter*.” The text is tiny, and due to the low-quality image, it is challenging to read without squinting a bit. That said, the MCS OCR API can still OCR the text (although the text at the bottom of the trashcan is illegible — neither human nor API could read that text).

The next sample image contains a national park sign shown in Figure 17.3:

```
$ python microsoft_ocr.py --image images/park.png
[INFO] making request to Microsoft Cognitive Services API...
```

PLEASE TAKE  
NOTHING BUT  
PICTURES  
LEAVE NOTHING  
BUT FOOT PRINTS



**Figure 17.3.** OCR'ing a park sign using the Microsoft Cognitive Services OCR API. Notice that API can give us *rotated* text bounding boxes along with the OCR'd text itself.

The MCS OCR API can OCR each sign, line-by-line (Figure 17.3). Note that we're also able to compute rotated text bounding box/polygons for each line.

The final example we have contains traffic signs:

```
$ python microsoft_ocr.py --image images/street_signs.png
[INFO] making request to Microsoft Cognitive Services API...
```

Old Town Rd

STOP  
ALL WAY

Figure 17.4 shows that we can correctly OCR each piece of text on both the stop sign and street name sign.



Figure 17.4. The Microsoft Cognitive Services OCR API can detect the text on traffic signs.

### 17.3 Summary

In this chapter, you learned about Microsoft Cognitive Services (MCS) OCR API. Despite being slightly harder to implement and use than the Amazon Rekognition API, the Microsoft Cognitive Services OCR API demonstrated that it's quite robust and able to OCR text in many situations, *including* low-quality images.

When working with low-quality images, the MCS API *shined*. Typically, I would recommend that you programmatically detect and discard low-quality images (as we did in Chapter 13). However, if you find yourself in a situation where you *have* to work with low-quality images, it may be worth your while to use the Microsoft Azure Cognitive Services OCR API.



## Chapter 18

# Text Detection and OCR with Google Cloud Vision API

In our previous two chapters, we learned how to use the cloud-based optical character recognition (OCR) APIs, Amazon Rekognition API, and Microsoft Cognitive Services (MCS). Amazon Rekognition API required less code and was a bit more straightforward to implement; however, MCS showed its utility could correctly OCR low-quality images.

This chapter will look at one final cloud-based OCR service, the Google Cloud Vision API. In terms of code, the Google Cloud Vision API is easy to utilize. Still, it requires that we use their admin panel to generate a client JavaScript Object Notation (JSON) file that contains all the necessary information to access the Vision API.

I have mixed feelings about the JSON file. On the one hand, it's nice not to have to hardcode our private and public keys. But on the other hand, it's cumbersome to have to use the admin panel to generate the JSON file itself.

Realistically, it's a situation of "six of one, half a dozen of the other." It doesn't make that much of a difference (just something to be aware of).

And as we'll find out, the Google Cloud Vision API, just like the others, tends to be quite accurate and does a good job OCR'ing complex images.

Let's dive in!

### 18.1 Chapter Learning Objectives

In this chapter, you will:

- Learn how to obtain your Google Cloud Vision API keys/JSON configuration file from the Google cloud admin panel
- Configure your development environment for use with the Google Cloud Vision API
- Implement a Python script used to make requests to the Google Cloud Vision API

## 18.2 Google Cloud Vision API for OCR

In the first part of this chapter, you'll learn about the Google Cloud Vision API and how to obtain your API keys and generate your JSON configuration file for authentication with the API.

From there, we'll be sure to have your development environment correctly configured with the required Python packages to interface with the Google Cloud Vision API.

We'll then implement a Python script that takes an input image, packages it within an API request, and sends it to the Google Cloud Vision API for OCR.

We'll wrap up this chapter with a discussion of our results.

### 18.2.1 Obtaining Your Google Cloud Vision API Keys

#### 18.2.1.1 Prerequisite

A Google Cloud account with billing enabled is all you'll need to use the Google Cloud Vision API. You can find the Google Cloud guide on how to modify your billing settings here: <http://pyimg.co/y0a0d>.

#### 18.2.1.2 Steps to Enable Google Cloud Vision API and Download Credentials

You can find our guide to getting your keys at the following site: <http://pyimg.co/erftn>. These keys are required if you want to follow this chapter or use the Google Cloud Vision API in your projects.

### 18.2.2 Configuring Your Development Environment for the Google Cloud Vision API

If you have not already installed the `google-cloud-vision` Python package in your Python development environment, take a second to do so now:

---

```
$ pip install --upgrade google-cloud-vision
```

---

If you are using a Python virtual environment or an Anaconda package manager, be sure to use the appropriate command to access your Python environment *before* running the above pip-install command. Otherwise, the `google-cloud-vision` package will be installed in your system Python rather than your Python environment.

### 18.2.3 Project Structure

Let's inspect the project directory structure for our Google Cloud Vision API OCR project:

---

```
|-- images
|   |-- aircraft.png
|   |-- challenging.png
|   |-- street_signs.png
|-- client_id.json
|-- google_ocr.py
```

---

We will apply our `google_ocr.py` script to several examples in the `images` directory.

The `client_id.json` file provides all necessary credentials and authentication information. The `google_ocr.py` script will load this file and supply it to the Google Cloud Vision API to perform OCR.

### 18.2.4 Implementing the Google Cloud Vision API Script

With our project directory structure reviewed, we can move on to implementing `google_ocr.py`, the Python script responsible for:

- i. Loading the contents of our `client_id.json` file
- ii. Connecting to the Google Cloud Vision API
- iii. Loading and submitting our input image to the API
- iv. Retrieving the text detection and OCR results
- v. Drawing and displaying the OCR'd text to our screen

Let's dive in:

---

```

1 # import the necessary packages
2 from google.oauth2 import service_account
3 from google.cloud import vision
4 import argparse
5 import cv2
6 import io

```

---

**Lines 2–6** import our required Python packages. Note that we need the `service_account` to connect to the Google Cloud Vision API while the `vision` package contains the `text_detection` function responsible for OCR.

Next, we have `draw_ocr_results`, a convenience function used to annotate our output image:

---

```

8 def draw_ocr_results(image, text, rect, color=(0, 255, 0)):
9     # unpacking the bounding box rectangle and draw a bounding box
10    # surrounding the text along with the OCR'd text itself
11    (startX, startY, endX, endY) = rect
12    cv2.rectangle(image, (startX, startY), (endX, endY), color, 2)
13    cv2.putText(image, text, (startX, startY - 10),
14                cv2.FONT_HERSHEY_SIMPLEX, 0.8, color, 2)
15
16    # return the output image
17    return image

```

---

The `draw_ocr_results` function accepts four parameters:

- i. `image`: The input image we are drawing on
- ii. `text`: The OCR'd text
- iii. `rect`: The bounding box coordinates of the text ROI
- iv. `color`: The color of the drawn bounding box and text

**Line 11** unpacks the  $(x, y)$ -coordinates of our text ROI. We use these coordinates to draw a bounding box surrounding the text along with the OCR'd text itself (**Lines 12–14**).

We then return the `image` to the calling function.

Let's examine our command line arguments:

---

```

19 # construct the argument parser and parse the arguments
20 ap = argparse.ArgumentParser()
21 ap.add_argument("-i", "--image", required=True,

```

---

---

```

22     help="path to input image that we'll submit to Google Vision API")
23 ap.add_argument("-c", "--client", required=True,
24     help="path to input client ID JSON configuration file")
25 args = vars(ap.parse_args())

```

---

We have two command line arguments here:

- **--image**: The path to the input image that we'll be submitting to the Google Cloud Vision API for OCR.
- **--client**: The client ID JSON file containing our authentication information (be sure to follow Section 18.2.1 to generate this JSON file).

It's time to connect to the Google Cloud Vision API:

---

```

27 # create the client interface to access the Google Cloud Vision API
28 credentials = service_account.Credentials.from_service_account_file(
29     filename=args["client"],
30     scopes=["https://www.googleapis.com/auth/cloud-platform"])
31 client = vision.ImageAnnotatorClient(credentials=credentials)
32
33 # load the input image as a raw binary file (this file will be
34 # submitted to the Google Cloud Vision API)
35 with io.open(args["image"], "rb") as f:
36     byteImage = f.read()

```

---

**Lines 28–30** connect to the Google Cloud Vision API, supplying the `--client` path to the JSON authentication file on disk. **Line 31** then creates our `client` for all image processing/computer vision operations.

We then load our input `--image` from disk as a byte array (`byteImage`) to submit it to Google Cloud Vision API.

Let's submit our `byteImage` to the API now:

---

```

38 # create an image object from the binary file and then make a request
39 # to the Google Cloud Vision API to OCR the input image
40 print("[INFO] making request to Google Cloud Vision API...")
41 image = vision.Image(content=byteImage)
42 response = client.text_detection(image=image)
43
44 # check to see if there was an error when making a request to the API
45 if response.error.message:
46     raise Exception(
47         "{}\nFor more info on errors, check:\n{}".format(
48             response.error.message))

```

---

**Line 41** creates an `Image` data object, which is then submitted to the `text_detection` function of the Google Cloud Vision API (**Line 42**).

**Lines 45–49** check to see if there was an error OCR'ing our input image and if so, we raise the error and exit from the script.

Otherwise, we can process the results of the OCR step:

---

```

51 # read the image again, this time in OpenCV format and make a copy of
52 # the input image for final output
53 image = cv2.imread(args["image"])
54 final = image.copy()
55
56 # loop over the Google Cloud Vision API OCR results
57 for text in response.text_annotations[1::]:
58     # grab the OCR'd text and extract the bounding box coordinates of
59     # the text region
60     ocr = text.description
61     startX = text.bounding_poly.vertices[0].x
62     startY = text.bounding_poly.vertices[0].y
63     endX = text.bounding_poly.vertices[1].x
64     endY = text.bounding_poly.vertices[2].y
65
66     # construct a bounding box rectangle from the box coordinates
67     rect = (startX, startY, endX, endY)

```

---

**Line 53** loads our input image from disk in OpenCV/NumPy array format (so that we can draw on it).

**Line 57** loops over all OCR'd `text` from the Google Cloud Vision API response. **Line 60** extracts the `ocr` text itself, while **Lines 61–64** extract the text region's bounding box coordinates. **Line 67** then constructs a rectangle (`rect`) from these coordinates.

The final step is to draw the OCR results on the `output` and `final` images:

---

```

69 # draw the output OCR line-by-line
70 output = image.copy()
71 output = draw_ocr_results(output, ocr, rect)
72 final = draw_ocr_results(final, ocr, rect)
73
74 # show the output OCR'd line
75 print(ocr)
76 cv2.imshow("Output", output)
77 cv2.waitKey(0)
78
79 # show the final output image
80 cv2.imshow("Final Output", final)
81 cv2.waitKey(0)

```

---

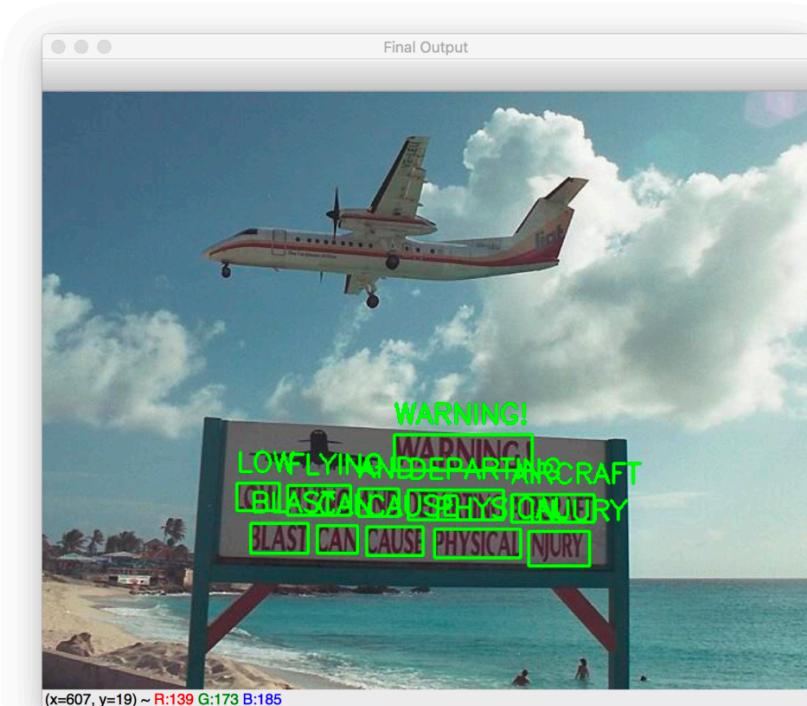
Each piece of OCR'd text is displayed on our screen on **Lines 75–77**. The final image, with *all* OCR'd text, is displayed on **Lines 80 and 81**.

### 18.2.5 Google Cloud Vision API OCR Results

Let's now put the Google Cloud Vision API to work! Open a terminal and execute the following command:

```
$ python google_ocr.py --image images/aircraft.png --client client_id.json
[INFO] making request to Google Cloud Vision API...
WARNING!
LOW
FLYING
AND
DEPARTING
AIRCRAFT
BLAST
CAN
CAUSE
PHYSICAL
INJURY
```

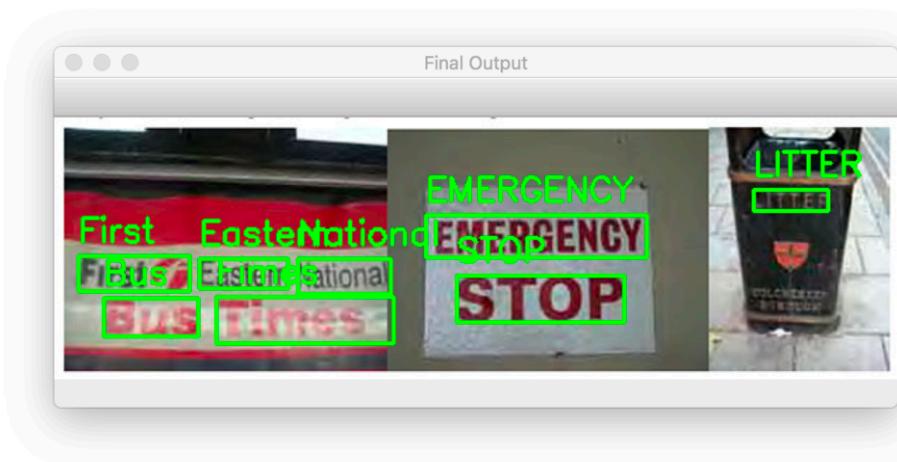
Figure 18.1 shows the results of applying the Google Cloud Vision API to our aircraft image, the same image we have been benchmarking OCR performance across all three cloud services. Like Amazon Rekognition API and Microsoft Cognitive Services, the Google Cloud Vision API can correctly OCR the image.



**Figure 18.1.** OCR'ing a warning sign line-by-line using the Google Cloud Vision API.

Let's try a more challenging image, which you can see in Figure 18.2:

```
$ python google_ocr.py --image images/challenging.png --client client_id.json
[INFO] making request to Google Cloud Vision API...
LITTER
First
Eastern
National
Bus
Times
EMERGENCY
STOP
```



**Figure 18.2.** OCR'ing a more challenging example with the Google Cloud Vision API. The OCR results are nearly 100% correct, the exception being that the “T” in “Times” is an “F.”

Just like the Microsoft Cognitive Services API, the Google Cloud Vision API performs well on our challenging, low-quality image with pixelation and low readability (even by human standards, let alone a machine). The results are in Figure 18.2.

Interestingly, the Google Cloud Vision API *does* make a mistake, thinking that the “T” in “Times” is an “F.”

Let's look at one final image, this one of a street sign:

```
$ python google_ocr.py --image images/street_signs.png --client client_id.json
[INFO] making request to Google Cloud Vision API...
Old
Town
Rd
STOP
ALL
WAY
```

Figure 18.3 displays the output of applying the Google Cloud Vision API to our street sign image. Microsoft Cognitive Services API OCRs the image line-by-line, resulting in the text “*Old Town Rd*” and “*All Way*” to be OCR’d as a single line. Alternatively, Google Cloud Vision API OCRs the text word-by-word (the default setting in the Google Cloud Vision API).



**Figure 18.3.** The Google Cloud Vision API OCRs our street signs but, by default, returns the results word-by-word.

### 18.3 Summary

In this chapter, you learned how to utilize the cloud-based Google Cloud Vision API for OCR. Like the other cloud-based OCR APIs we’ve covered in this book, the Google Cloud Vision API can obtain high OCR accuracy with little effort. The downside, of course, is that you need an internet connection to leverage the API.

When choosing a cloud-based API, I wouldn’t focus on the amount of Python code required to interface with the API. **Instead, consider the *overall ecosystem* of the cloud platform you are using.**

Suppose you’re building an application that requires you to interface with Amazon Simple Storage Service (Amazon S3) for data storage. In that case, it makes a lot more sense to use Amazon Rekognition API. This enables you to keep everything under the Amazon umbrella.

On the other hand, if you are using the Google Cloud Platform (GCP) instances to train deep learning models in the cloud, it makes more sense to use the Google Cloud Vision API.

These are all design and architectural decisions when building your application. Suppose you're just "testing the waters" of each of these APIs. You are not bound to these considerations. However, if you're developing a production-level application, then it's *well worth your time* to consider the trade-offs of each cloud service. You should consider *more* than just OCR accuracy; consider the compute, storage, etc., services that each cloud platform offers.

## Chapter 19

# Training a Custom Tesseract Model

So far, in this text, we have learned how to create OCR models by:

- Leveraging basic computer vision and image processing techniques to detect and extract characters
- Utilizing OpenCV to recognize extracted characters
- Training custom Keras/TensorFlow models
- Using Tesseract to detect and recognize text within images

**However, we have yet to train/fine-tune a Tesseract model on our *custom dataset*.**

This chapter, along with the next, takes us deep down the rabbit hole into the world of Tesseract training commands, tools, and proper dataset/project directory structures.

As you'll see, training a custom Tesseract model has absolutely *nothing* to do with programming and writing code and instead involves:

- i. Properly configuring your development environment with the required Tesseract training tools
- ii. Structuring your OCR dataset in a manner Tesseract can understand
- iii. Correctly setting Tesseract environment variables in your shell
- iv. Running the correct Tesseract training commands

Suppose you've ever worked with Caffe [59] or the TensorFlow Object Detection API [60], then you know these tools typically don't require you to open an integrated development environment (IDE) and write hundreds of lines of code.

Instead, they are *far* more focused on ensuring the dataset is correctly structured, followed by issuing a series of (sometimes complicated) commands.

**Training a custom Tesseract model is the same** — and in the remainder of this chapter, you will learn how to fine-tune a Tesseract model on a custom dataset.

## 19.1 Chapter Learning Objectives

In this chapter, you will learn how to:

- i. Configure your development environment for Tesseract training
- ii. Install the `tesstrain` package, which utilizes `make` files to construct training commands and commence training easily
- iii. Understand the dataset directory structure required when training a custom Tesseract model
- iv. Review the dataset we'll be using when training our Tesseract model
- v. Configure the base Tesseract model that we'll be training
- vi. Train the actual Tesseract OCR model on our custom dataset

## 19.2 Your Tesseract Training Development Environment

In the first part of this section, we'll configure our system with the required packages to train a custom Tesseract model.

Next, we'll compile Tesseract from source. **Compiling Tesseract from source is a requirement when training a custom Tesseract OCR model.** The `tesseract` binary installed via `apt-get` or `brew` will *not* be sufficient.

Finally, we'll install `tesstrain`, a set of Python and makefile utilities to train the custom Tesseract model.

If possible, I suggest using a virtual machine (VM) or a cloud-based instance the first time you configure a system to train a Tesseract model. The reasoning behind this suggestion is simple — you *will* make a mistake once, twice, and probably even three times before you properly configure your environment.

Configuring Tesseract to train a custom model for the first time is *hard*, so having an environment where you can snapshot a base operating system install, do your work with

Tesseract, and when something goes wrong, instead of having to reconfigure your bare metal machine, you can instead revert to your previous state.

I genuinely hope you take my advice — this process is not for the faint of heart. You need extensive Unix/Linux command line experience to train custom Tesseract models due to the very nature that *everything* we are doing in this chapter involves the command line.

If you are new to the command line, or you don't have much experience working with package managers such as `apt-get` (Linux) or `brew` (macOS), I suggest you table this chapter for a bit while you improve your command line skills.

With all that said, let's start configuring our development environment such that we can train custom Tesseract models.

### 19.2.1 Step #1: Installing Required Packages

This section provides instructions to install operating-system-level packages required to (1) compile Tesseract from source and (2) train custom OCR models with Tesseract.

Unfortunately, compiling Tesseract from source is a *requirement* to use the training utilities.

Luckily, installing our required packages is as simple as a few `apt-get` and `brew` commands.

#### 19.2.1.1 Ubuntu

The following set of commands installs various C++ libraries, compiles (`g++`), package managers (`git`), image I/O utilities (so we can load images from disk), and other tools required to compile Tesseract from source:

---

```
$ sudo apt-get install libicu-dev libpango1.0-dev libcairo2-dev
$ sudo apt-get install automake ca-certificates g++ git libtool libleptonica-dev
→ make pkg-config
$ sudo apt-get install --no-install-recommends asciidoc docbook-xsl xsltproc
$ sudo apt-get install libpng-dev libjpeg8-dev libtiff5-dev zlib1g-dev
```

---

Assuming none of these commands error out, your Ubuntu machine should have the proper OS-level packages installed.

#### 19.2.1.2 macOS

If you're on a macOS machine, installing the required OS-level packages is *almost* as easy — all we need is to use Homebrew (<https://brew.sh> [61]) and the `brew` command:

---

```
$ brew install icu4c pango cairo
$ brew install automake gcc git libtool leptonica make pkg-config
$ brew install libpng jpeg libtiff zlib
```

---

However, there is an additional step required. The `icu4c` package, which contains C/C++ and Java libraries for working with Unicode, requires that we *manually* create a symbolic link (sym-link) for it into our `/usr/local/opt` directory:

---

```
$ ln -hfs /usr/local/Cellar/icu4c/67.1 /usr/local/opt/icu4c
```

---

Note that the current version (as of this writing) for `icu4c` is 67.1. I suggest you use tab completion when running the command above. New versions of `icu4c` will require the command to be updated to utilize the correct version (otherwise, the path will be invalid, and thus the sym-link will be invalid as well).

### 19.2.2 Step #2: Building and Installing Tesseract Training Tools

Now that our OS-level packages are installed, we can move on to compiling Tesseract from source, giving us access to Tesseract's training utilities.

#### 19.2.2.1 Ubuntu

On an Ubuntu machine, you can use the following set of commands to:

- i. Download Tesseract v4.1.1
- ii. Configure the build
- iii. Compile Tesseract
- iv. Install Tesseract and its training utilities

The commands follow:

---

```
$ wget -O tesseract.zip https://github.com/tesseract-ocr/tesseract/archive/4.1.1.zip
$ unzip tesseract.zip
$ mv tesseract-4.1.1 tesseract
$ cd tesseract
$ ./autogen.sh
$ ./configure
$ make
$ sudo make install
```

---

```
$ sudo ldconfig
$ make training
$ sudo make training-install
```

---

Provided that you configured your Ubuntu development environment correctly in Section 19.2.1.1, you shouldn't have any issues compiling Tesseract from source.

However, if you run into an error message, don't worry! All hope is not lost yet.

Keep in mind what I said in this chapter's introduction — training a custom Tesseract model is *hard work*. The majority of the effort involves *configuring* your development environment. Once your development environment is configured correctly, training your Tesseract model becomes a breeze.

Tesseract is the world's most popular open-source OCR engine. Someone else has likely encountered the *same* error as you have. Copy and paste your error message in Google and start reading GitHub Issues threads. It may take some time and patience, but you will eventually find the solution.

Lastly, we will be downloading the English language and OSD Tesseract model files that would otherwise be installed automatically if you were installing Tesseract using `apt-get`:

---

```
$ cd /usr/local/share/tessdata/
$ sudo wget
→ https://github.com/tesseract-ocr/tessdata_fast/raw/master/eng.traineddata
$ sudo wget
→ https://github.com/tesseract-ocr/tessdata_fast/raw/master/osd.traineddata
```

---

### 19.2.2.2 macOS

Installing Tesseract on macOS has a near identical set of commands as our Ubuntu install:

---

```
$ wget -O tesseract.zip https://github.com/tesseract-ocr/tesseract/archive/4.1.1.zip
$ unzip tesseract.zip
$ mv tesseract-4.1.1 tesseract
$ cd tesseract
$ ./autogen.sh
$ export PKG_CONFIG_PATH="/usr/local/opt/icu4c/lib/pkgconfig"
$ ./configure
$ make
$ sudo make install
$ make training
$ sudo make training-install
```

---

The big difference here is that we need to explicitly set our `PKG_CONFIG_PATH` environment variable to point to the `icu4c` sym-link we created in Section 19.2.1.2.

**Do not forget this `export` command — if you do, your Tesseract compile will error out.**

Lastly, we will be downloading the English language and OSD Tesseract model files that would otherwise be installed automatically if you were installing Tesseract using `brew`:

---

```
$ cd /usr/local/share/tessdata/
$ sudo wget
→ https://github.com/tesseract-ocr/tessdata_fast/raw/master/eng.traineddata
$ sudo wget
→ https://github.com/tesseract-ocr/tessdata_fast/raw/master/osd.traineddata
```

---

### 19.2.3 Step #3: Cloning `tesstrain` and Installing Requirements

At this point, you should have Tesseract v4.1.1, compiled from source, installed on your system.

The next step is installing `tesstrain`, a set of Python tools that allow us to work with `make` files to train custom Tesseract models.

Using `tesstrain` (<http://pyimg.co/n6pdk> [62]), we can train and fine-tune Tesseract's deep learning-based LSTM models (the same model we've used to obtain high accuracy OCR results throughout this text).

A detailed review of the LSTM architecture is outside the scope of this text, but if you need a refresher, I recommend this fantastic intro tutorial by Christopher Olah (<http://pyimg.co/9hmte> [63]).

Perhaps most importantly, the `tesstrain` package provides instructions on how our training dataset should be structured, making it *far easier* to build our dataset and train the model.

Let's install the required Python packages required by `tesstrain` now:

---

```
$ cd ~
$ git clone https://github.com/tesseract-ocr/tesstrain
$ workon ocr
$ cd tesstrain
$ pip install -r requirements.txt
```

---

As you can see, installing `tesstrain` follows a basic `setup.py install`. All we have to do is clone the repository, change directory, and install the packages.

I am using the `workon` command to access my `ocr` Python virtual environment, thus keeping my Tesseract training Python libraries *separate* from my system Python packages.

Using Python virtual environments is the best practice and one I recommend you use. Still, if

you want to install these packages into your system Python install, that's okay as well (but again, *not recommended*).

## 19.3 Training Your Custom Tesseract Model

Take a second to congratulate yourself! You've done a lot of work configuring your Tesseract development environment — it was not easy, but you fought through. And the good news is, it's essentially all downhill from here!

In the first part of this section, we'll review our project directory structure. We'll then review the dataset we'll be using for this chapter. I'll also provide tips and suggestions for building your Tesseract training dataset.

Next, I'll show you how to download a pre-trained Tesseract model that we'll fine-tune on our dataset.

Once the model is downloaded, all that's left is to set a few environment variables, copy our training dataset into the proper directory, and then commence training!

### 19.3.1 Project Structure

Now that Tesseract and `tesstrain` are configured, we can move on to defining our directory structure for our project:

---

```
|-- training_data
|   |-- 01.gt.txt
|   |-- 01.png
|   |-- 02.gt.txt
|   |-- 02.png
...
|   |-- 18.gt.txt
|   |-- 18.png
|-- training_setup.sh
```

---

Our `training_data` directory is arguably the most important part of this project structure for you to understand. Inside this directory, you'll see that we have pairs of `.txt` and `.png` files.

The `.png` files are our images, containing the text in image format. The `.txt` files then have the plaintext version of the text, such that a computer can read it from disk.

When training, Tesseract will load each `.png` image and `.txt` file, pass it through the LSTM, and then update the model's weights, ensuring that it makes better predictions during the next batch.

**Most importantly, notice the naming conventions of the files in `training_data`.** For each `.png` file, you also have a text file named `xx.gt.txt`. For each image file, you *must* have a text file with the same filename, but instead of a `.png`, `.jpg`, etc., extension, it *must* have a `.gt.txt` extension. Tesseract can associate the images and ground-truth (`gt`) text files together, but you must have the correct file names and extensions.

We'll cover the actual dataset itself in the next section, but simply understand the naming convention for the time being.

We then have a shell script, `training_setup.sh`. This shell script handles setting an important environment variable, `TESSDATA_PREFIX`, which allows us to perform training and inference.

### 19.3.2 Our Tesseract Training Dataset

Our Tesseract training dataset consists of 18 lines of random, handwritten words (Figure 19.1). This dataset was put together by PyImageSearch reader Jayanth Rasamsetti, who helped us put together all the pieces for this chapter.

To construct our training set, we manually cropped each row of text, creating 18 separate images. We cropped each row using basic image editors (e.g., macOS Preview, Microsoft Paint, GIMP, Photoshop, etc.).

If wandered relation no surprise

of screened doubtful. Overcame no

instead ye of trifling husbands.

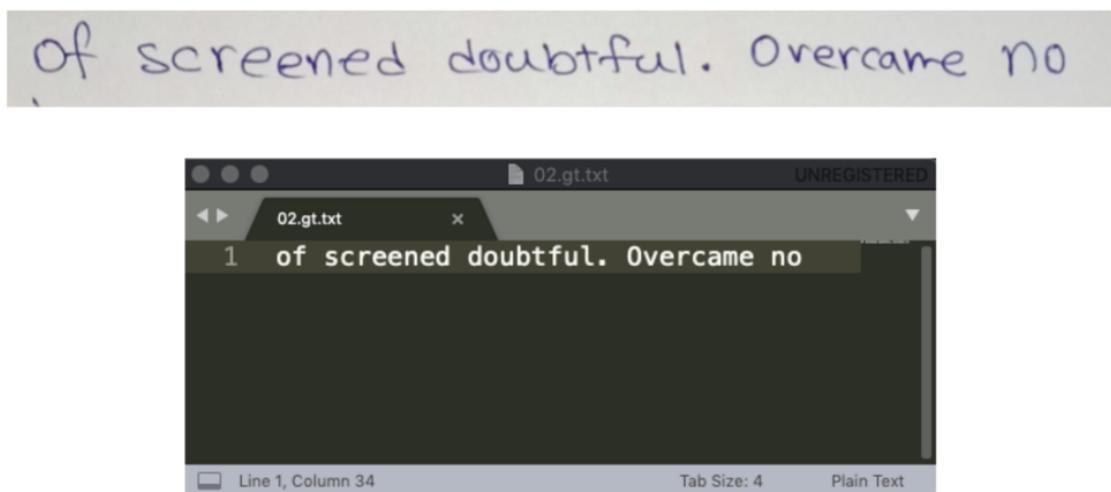
Might am older hours on found.

**Figure 19.1.** A few images of handwritten text from our training dataset.

Then, for each row, we created a `.txt` file containing the ground-truth text. For example, let's examine the second row (`02.png`) and its corresponding `02.gt.txt` file (Figure 19.2).

As Figure 19.2 displays, we have a row of text in an image (*top*), while the *bottom* contains the ground-truth text as a plaintext file.

We then have the same file structure for all 18 rows of text. We'll use this data to train our custom Tesseract model.



**Figure 19.2.** *Top:* A sample image from our training dataset. *Bottom:* Ground-truth label of the sample image in `.gt.txt` format.

### 19.3.3 Creating Your Tesseract Training Dataset

You might be wondering why we are using Tesseract to train a custom handwriting recognition model. Astute OCR practitioners will know that Tesseract doesn't perform well on handwriting recognition compared to typed text. *Why bother training a handwriting recognition model at all?*

It boils down to three reasons:

- i. **Readers ask all the time if it's possible to train a Tesseract model for handwriting recognition.** As you'll see, it's possible, and our custom model's accuracy will be  $\approx 10x$  better than the default model, but it still leaves accuracy to be desired.
- ii. **Training a custom Tesseract model for typed text recognition is comparatively “easy.”** Our goal in this chapter is to show you something you haven't seen before.
- iii. **Most importantly, once you've trained a custom Tesseract model, you can train the model on your data.** I suggest replicating the results of this chapter first to get a feel for

the training process. From there, training your model is as easy as swapping the `training_data` directory for your images and `.gt.txt` files.

#### 19.3.4 Step #1: Setup the Base Model

Instead of training a Tesseract model from scratch, which is time-consuming, tedious, and harder to do, we're going to *fine-tune* an existing LSTM-based model.

The LSTM models included with Tesseract v4 are *much more accurate* than the non-deep learning models in Tesseract v3, so I suggest you use these LSTM models and fine-tune them rather than training from scratch.

Additionally, it's worth mentioning here that Tesseract has three different types of model types:

- i. `tessdata_fast`: These models balance speed vs. accuracy using integer-quantized weights. These are also the default models that are included when you install Tesseract on your machine.
- ii. `tessdata_best`: These models are the most accurate, using floating-point data types.  
**When fine-tuning a Tesseract model, you *must* start with these models.**
- iii. `tessdata`: The old legacy OCR models from 2016. We will not need or utilize these models.

You can learn more about each of these three model types in Tesseract's documentation (<http://pyimg.co/frg2n> [64]).

You can use the following commands to change directory to `~/tesseract/tessdata` and then download the `eng.traineddata` file, which is the trained LSTM model for the English language:

---

```
$ cd ~/tesseract
$ cd tessdata
$ wget
→ https://raw.githubusercontent.com/tesseract-ocr/tessdata_best/master/
→ eng.traineddata
```

---

Tesseract maintains a repository of all `.traineddata` model files for 124 languages here: <http://pyimg.co/7n22b>.

We use the English language model in this chapter. Still, you can just as easily replace the English language model with a language of your choosing (provided there is a `.traineddata` model for that language, of course).

### 19.3.5 Step #2: Set Your TESSDATA\_PREFIX

With our `.traineddata` model downloaded (i.e., the weights of the model that we'll be fine-tuning), we now need to set an important environment variable `TESSDATA_PREFIX`.

The `TESSDATA_PREFIX` variable needs to point to our `tesseract/tessdata` directory.

If you followed the install instructions (Section 19.2.2), where we compiled Tesseract and its training tools from scratch, then the `tesseract/tessdata` folder should be in your home directory. **Take a second now to verify the location of `tesseract/tessdata` — double-check and triple-check this file path; setting the incorrect `TESSDATA_PREFIX` will result in Tesseract being unable to find your training data and thus unable to train your model.**

Inside the project directory structure for this chapter, you will find a file named `training_setup.sh`. As the name suggests, this shell script sets your `TESSDATA_PREFIX`.

Let's open this file and inspect the contents now:

---

```
#!/bin/sh

export TESSDATA_PREFIX="/home/pyimagesearch/tesseract/tessdata"
```

---

Here you can see that I've supplied the *absolute path* to my `tesseract/tessdata` directory on disk. You *need* to supply the absolute path; otherwise, the training command will error out.

Additionally, my username on this machine is `pyimagesearch`. If you use the VM included with your purchase of this text, this VM *already* has Tesseract and its training tools installed. **If your username is different, you will need to update the `training_setup.sh` file and replace the `pyimagesearch` text with your username.**

To set the `TESSDATA_PREFIX` environment variable, you can run the following command:

---

```
$ source training_setup.sh
```

---

As a sanity check, let's print the contents of the `TESSDATA_PREFIX` variable to our terminal:

---

```
$ echo $TESSDATA_PREFIX
/home/pyimagesearch/tesseract/tessdata
```

---

The final step is to verify the `tesseract/tessdata` directory exists on disk by copying and pasting the above path into the `ls` command:

---

```
$ ls -l /home/pyimagesearch/tesseract/tessdata
total 15112
-rw-rw-r-- 1 ubuntu ubuntu    22456 Nov 24 08:29 Makefile
-rw-rw-r-- 1 ubuntu ubuntu      184 Dec 26 2019 Makefile.am
-rw-rw-r-- 1 ubuntu ubuntu   22008 Nov 24 08:29 Makefile.in
drwxrwxr-x 2 ubuntu ubuntu     4096 Nov 24 08:29 configs
-rw-rw-r-- 1 ubuntu ubuntu 15400601 Nov 24 08:41 eng.traineddata
-rw-rw-r-- 1 ubuntu ubuntu      33 Dec 26 2019 eng.user-patterns
-rw-rw-r-- 1 ubuntu ubuntu     27 Dec 26 2019 eng.user-words
-rw-rw-r-- 1 ubuntu ubuntu     572 Dec 26 2019 pdf.ttf
drwxrwxr-x 2 ubuntu ubuntu     4096 Nov 24 08:29 tessconfigs
```

---

If your output looks similar to mine, then you know your `TESSDATA_PREFIX` variable is set correctly.

If your `TESSDATA_PREFIX` variable is set *incorrectly*, then `ls` will report an error:

---

```
$ ls -l /home/pyimagesearch/tesseract/tessdata
ls: /home/pyimagesearch/tesseract/tessdata: No such file or directory
```

---

This error implies that your path to `tesseract/tessdata` is incorrect. Go back and check your file paths. **You need to have `TESSDATA_PREFIX` correctly set before continuing!**

### 19.3.6 Step #3: Setup Your Training Data

In Section 19.3.1, we reviewed the directory structure and file organization Tesseract and `tesstrain` expect when training/fine-tuning a custom Tesseract OCR model. Section 19.3.3 then provided suggestions on how to build your training dataset.

At this point, I'll assume that you have your dataset properly organized on disk.

**For the sake of this example, I also suggest that you use the training data included in the directory structure of this project — walk before you run. Training a custom Tesseract model is hard, especially if this is your first time. I strongly encourage you to replicate our results before trying to train your models.**

The following commands copy our training data from the `training_data` directory of this project into `tesstrain/handwriting-ground-truth`:

---

```
$ cd ~/tesstrain
$ mkdir data
$ cd data
$ mkdir handwriting-ground-truth
$ cd handwriting-ground-truth
$ cp -r ~/OCRPractitionerBundle_Code/chapter19-training_tesseract_model/
→ training_data/*.* .
```

---

**The most important aspect of the above commands for you to understand is the *naming convention*.** Let's break each of these commands down into components to understand what we are doing:

First, a `cd ~/tesstrain` changes the directory into our base `tesstrain` directory. We then create a `data` directory and `cd` into it — we *must* create this `data` directory to store our training data.

Inside the `data` directory, we then create a subdirectory named `handwriting-ground-truth`. You **must understand the naming convention of this directory**:

- i. The `handwriting` portion of the subdirectory is *our model's name* (which we'll use in our next section during training). **You can name your model whatever you want**, but I suggest making it something specific to the project/task at hand, so you remember what the intended use of the model is.
- ii. The second portion of the directory name, `ground-truth`, tells Tesseract that this is where our training data lives.

**When creating your training data directory structure, you *must* follow this naming convention.**

After creating the `handwriting-ground-truth` directory and `cd` into it, we copy all of our images and `.txt` files into it.

Once that is complete, we can train the actual model.

### 19.3.7 Step #4: Fine-Tune the Tesseract Model

With our training data in place, we can now fine-tune our custom Tesseract model.

Open a terminal and execute the following commands:

---

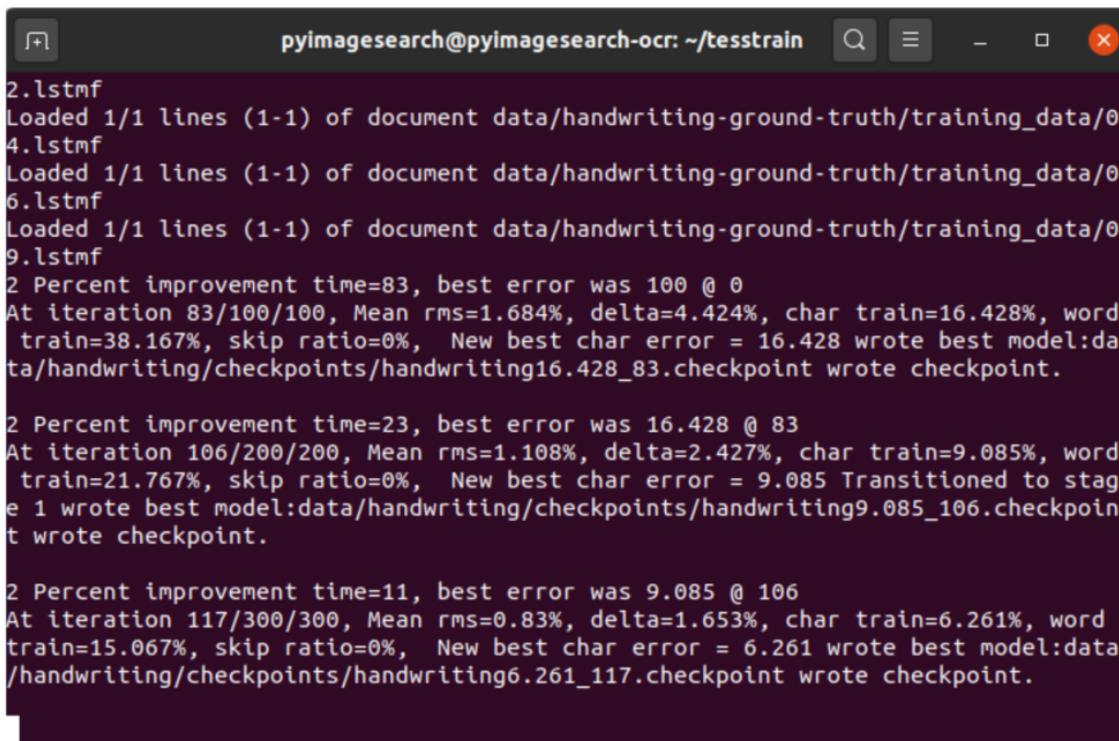
```
$ workon ocr
$ cd ~/tesstrain
$ make training MODEL_NAME=handwriting START_MODEL=eng
↪   TESSDATA=/home/pyimagesearch/tesseract/tessdata
```

---

First, I use the `workon` command to access the Python virtual environment where I have all my `tesstrain` dependencies installed. As I mentioned earlier in this chapter, using a Python virtual environment for these dependencies is *optional* but *suggested* (you should install the packages in the system Python, for instance).

Next, we change directory into `~/tesstrain`, where we have our Tesseract training utilities.

Training commences using the `make` command, the output of which you can see in Figure 19.3. Here you can see the model training. Training will continue until the default number of iterations, `MAX_ITERATIONS=10000`, is reached.



```
pyimagesearch@pyimagesearch-ocr: ~/tesstrain
2.lstmf
Loaded 1/1 lines (1-1) of document data/handwriting-ground-truth/training_data/0
4.lstmf
Loaded 1/1 lines (1-1) of document data/handwriting-ground-truth/training_data/0
6.lstmf
Loaded 1/1 lines (1-1) of document data/handwriting-ground-truth/training_data/0
9.lstmf
2 Percent improvement time=83, best error was 100 @ 0
At iteration 83/100/100, Mean rms=1.684%, delta=4.424%, char train=16.428%, word
train=38.167%, skip ratio=0%, New best char error = 16.428 wrote best model:da
ta/handwriting/checkpoints/handwriting16.428_83.checkpoint wrote checkpoint.

2 Percent improvement time=23, best error was 16.428 @ 83
At iteration 106/200/200, Mean rms=1.108%, delta=2.427%, char train=9.085%, word
train=21.767%, skip ratio=0%, New best char error = 9.085 Transitioned to stag
e 1 wrote best model:data/handwriting/checkpoints/handwriting9.085_106.checkpoi
nt wrote checkpoint.

2 Percent improvement time=11, best error was 9.085 @ 106
At iteration 117/300/300, Mean rms=0.83%, delta=1.653%, char train=6.261%, word
train=15.067%, skip ratio=0%, New best char error = 6.261 wrote best model:da
ta/handwriting/checkpoints/handwriting6.261_117.checkpoint wrote checkpoint.
```

**Figure 19.3.** Model training information after we successfully launch the training.

On my machine, the entire training process took  $\approx$ 10 minutes. The training logs are shown in Figure 19.4.

After the training is completed, the final logs should look like Figure 19.4.

But before we get too far, let's break down the `make` command to ensure we understand each of the parameters:

- i. `MODEL_NAME`: The name of the Tesseract model we are training. This value should be the *same* as the directory you created in `tesstrain/data`; that way, Tesseract knows to associate the `tesstrain/data/handwriting-ground-truth` dataset with the model we are training.
- ii. `START_MODEL`: The name of the model we are fine-tuning. Recall that back in Section 19.3.4, and we downloaded `eng.traineddata` to `tesseract/tessdata` — that file holds the initial model weights that we will be fine-tuning.

- iii. TESSDATA: The *absolute path* to our tesseract/tessdata directory where the handwriting subdirectory and training data are stored.

```
2 Percent improvement time=1, best error was 2.087 @ 118
At iteration 119/1200/1200, Mean rms=0.115%, delta=0.014%, char train=0.064%, word train=0.167%, skip ratio=0%,
New best char error = 0.064 wrote best model:data/handwriting/checkpoints/handwriting0.064_119.checkpoint wrote checkpoint.

2 Percent improvement time=1, best error was 2.087 @ 118
At iteration 119/1300/1300, Mean rms=0.095%, delta=0.003%, char train=0.003%, word train=0%, skip ratio=0%,
New best char error = 0.003 wrote best model:data/handwriting/checkpoints/handwriting0.003_119.checkpoint wrote checkpoint.

Finished! Error rate = 0.003
lstmtraining \
--stop_training \
--continue_from data/handwriting/checkpoints/handwriting_checkpoint \
--traineddata data/handwriting/handwriting.traineddata \
--model_output data/handwriting.traineddata
Loaded file data/handwriting/checkpoints/handwriting_checkpoint, unpacking...
```

Figure 19.4. Model training completion information.

The previous three parameters are *required* to fine-tune the Tesseract model; however, it's worth noting that there are another two important hyperparameters that you may want to consider fine-tuning:

- i. MAX\_ITERATIONS: The total number of training iterations (similar to how Caffe iterations default to 10000).
- ii. LEARNING\_RATE: The learning rate when fine-tuning our Tesseract model. This value defaults to 0.0001. I recommend keeping the learning rate a relatively low value. If you raise it too high, then the model's pre-trained weights may be broken down too quickly (i.e., before the model can take advantage of the pre-trained weights to learn patterns specific to the current task).
- iii. PSM: The default PSM mode to use when segmenting the characters from your training images. You should use the `tesseract` command to test your example images with varying PSM modes until you find the correct one for segmentation (see Chapter 11, *Improving OCR Results with Tesseract Options* of the “*Intro to OCR*” Bundle for more information on page segmentation modes).

I assume that as a deep learning practitioner, you understand the importance of fine-tuning both your iterations and learning rate — fine-tuning these hyperparameters is *crucial* in obtaining a higher accuracy OCR model.

A full list of hyperparameters available to you can be found on the GitHub repository for the `tesstrain` package (<http://pyimg.co/n6pdk>).

I'll wrap up this chapter by saying if your training command errors out and you need to start

again (e.g., you accidentally supplied the incorrect `TESSDATA` path), all you need do is run the following command:

---

```
$ make clean MODEL_NAME=handwriting
```

---

Running `make clean` allows you to restart the training process. Make sure you specify the `MODEL_NAME`. Otherwise, Tesseract will assume the default model name (which is `foo`).

**If you make training command errors out, I suggest you run `make clean` to clean up and allow Tesseract a fresh start to train your model.**

### 19.3.8 Training Tips and Suggestions

When training your custom Tesseract models, the first thing you need to do is set your expectations. **For your first time around, it's going to be hard, and the only way it will get easier is to suffer through it, learning from your mistakes.**

There's no "one easy trick." There's no shortcut. There's no quick and dirty hack. Instead, you need to put in the hours and the time.

**Secondly, accept that you'll run into errors.** It's inevitable. Don't fight it, and don't get discouraged when it happens. You'll miss/forget commands, and you may run into development environment issues specific to your machine.

**When that happens, Google is your friend.** Tesseract is the world's most popular OCR engine. It's more than likely that someone has encountered the same error as you have. Copy and paste the error from your terminal, search for it on Google, and do your due diligence.

Yes, that is time-consuming. Yes, it's a bit tedious. And yes, there will be more than one false-start along the way.

Do you know what I have to say about that?

**Good. It means you're learning.**

The notion that you can train a custom Tesseract model in your pajamas with a hot cup of tea and someone massaging your shoulders, relaxing you into the process, is false. Accept that it's going to suck from time to time — if you can't accept it, you won't be able to learn how to do it.

**Training a custom Tesseract model comes with experience. You have to earn it first.**

Finally, if you run into an error that no one else has documented online, post it on the `tesstrain` GitHub Issues page (<http://pyimg.co/f15kp>) so that the Tesseract developers can

help you out. The Tesseract team is incredibly active and responds to bug reports and questions — utilize their knowledge to better yourself.

And above all, take your time and be patient. I know it can be frustrating. But you got this — just one step at a time.

## 19.4 Summary

In this chapter, you learned how to train and fine-tune a Tesseract OCR model on your custom dataset.

As you learned, fine-tuning a Tesseract model on your dataset has *very little* to do with writing code, and instead is all about:

- i. Properly configuring your development environment
- ii. Installing the `tesstrain` package
- iii. Structuring your OCR training dataset in a manner Tesseract can understand
- iv. Setting the proper environment variables correctly
- v. Running the correct training commands (and in the right order)

If you've ever worked with Caffe or the TensorFlow Object Detection API, then you know that working with commands, rather than code, can be a catch-22:

- On the one hand, it's *easier* since you aren't writing code, and therefore don't have to worry about bugs, errors, logic issues, etc.
- But on the other hand, it's *far harder* because you are left at the mercy of the Tesseract training tools

**Instead, I suggest you remove the word “easy” from your vocabulary when training a Tesseract model — it’s going to be *hard* — plain and simple.**

You need to adjust your mindset accordingly and walk in with an open mind, knowing that:

- The first (and probably the second and third) time you try to configure your development environment for Tesseract, training will likely fail due to missed commands, environment-specific error messages, etc.
- You'll need to double-check and triple-check that your training data is stored correctly in the `tessdata` directory

- Your training commands *will* error out if even the slightest configuration is correct
- Additionally, the Tesseract training command will error out because you forgot to be in a Python environment with the Python Imaging Library (PIL)/Pillow installed
- You'll forget to run `make clean` when running the training command and then won't be able to get training to start again (and you may spend an hour or so banging your head against the wall before you realize it)

**Accept that this is all part of the process.** It takes time and *a lot* of patience to get used to training custom Tesseract models, but the more you do it, the *easier* it gets.

In the following chapter, we will take our trained custom Tesseract model and OCR an input image.

## Chapter 20

# OCR’ing Text with Your Custom Tesseract Model

In our previous chapter, you learned how to train a Tesseract model on your custom dataset.

We did so by annotating a sample set of images, providing their ground-truth plaintext versions, configuring a Tesseract training job, and then fine-tuning Tesseract’s LSTM OCR model on our custom dataset.

After training is completed, we were left with a serialized OCR model. However, the question remains:

*“How do we take this trained Tesseract model and use it to OCR input images?”*

The remainder of this chapter will address that exact question.

### 20.1 Chapter Learning Objectives

Inside this chapter, you will:

- Learn how to set your `TESSDATA_PREFIX` path for inference
- Load our custom trained Tesseract OCR model
- Use the model to OCR input images

### 20.2 OCR with Your Custom Tesseract Model

In the first part of this chapter, we’ll review our project directory structure. We’ll then implement a Python script to take our custom trained OCR model and then use it to OCR an

input image. We'll wrap up the chapter by comparing our *custom model's accuracy* with the *default Tesseract model*, noting that our custom model can obtain higher OCR accuracy.

### 20.2.1 Project Structure

Let's start this chapter by reviewing our project directory structure:

---

```
|-- full.png
|-- full.txt
|-- handwriting.traineddata
|-- handwritten_to_text.py
|-- inference_setup.sh
```

---

Here's what you need to know about the directory structure:

- The `inference_setup.sh` script is used to set our `TESSDATA_PREFIX` environment variable for testing
- The `handwritten_to_text.py` file takes an input image (`full.png`), applies OCR to it, and then compares it to the ground-truth text (`full.txt`)
- The `handwriting.traineddata` is our custom trained Tesseract model serialized to disk (I included it here in the project structure just in case you do not wish to train the model yourself and instead just want to perform inference)

By the end of this chapter, you'll be able to take your custom trained Tesseract model and use it to OCR input images.

### 20.2.2 Implementing Your Tesseract OCR Script

With our custom Tesseract model trained, all that's left is to learn how to implement a Python script to load the model and apply it to a sample image.

Open the `handwritten_to_text.py` file in your project directory structure, and let's get to work:

---

```
1 # import the necessary packages
2 from difflib import SequenceMatcher as SQ
3 import pytesseract
4 import argparse
5 import cv2
```

---

**Lines 2–5** import our required Python packages. The `difflib` module provides functions for comparing sequences of files, strings, etc.

We'll specifically use the `SequenceMatcher`, which can accept a pair of input strings (i.e., the ground-truth text and the predicted OCR'd text) and then compare the sequences for overlap, resulting in an accuracy percentage.

The `pytesseract` package will be used to interface with the Tesseract OCR engine, and more specifically, to set the name of our custom trained Tesseract model.

Let's now move on to our command line arguments:

---

```

7 # construct the argument parser and parse the arguments
8 ap = argparse.ArgumentParser()
9 ap.add_argument("-n", "--name", required=True,
10                 help="name of the OCR model")
11 ap.add_argument("-i", "--image", required=True,
12                 help="path to input image of handwritten text")
13 ap.add_argument("-g", "--ground-truth", required=True,
14                 help="path to text file containing the ground-truth labels")
15 args = vars(ap.parse_args())

```

---

Our script requires three command line arguments, including:

- i. `--name`: The name of our OCR model. In the previous chapter, our output, a serialized OCR model, had the filename `handwriting.traineddata`, so the `--name` of the model is simply `handwriting` (i.e., leaving out the file extension).
- ii. `--image`: The path to the input image we want to which our custom OCR model applies.
- iii. `--ground-truth`: The plaintext file containing the ground-truth text in the input image; we'll compare the ground-truth text to the text produced by our custom OCR model.

With our command line arguments taken care of, let's proceed to OCR the input image:

---

```

17 # load the input image and convert it from BGR to RGB channel
18 # ordering
19 image = cv2.imread(args["image"])
20 image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
21
22 # use Tesseract to OCR the image
23 print("[INFO] OCR'ing the image...")
24 predictedText = pytesseract.image_to_string(image, lang=args["name"])
25 print(predictedText)

```

---

**Lines 19 and 20** load our input --image from disk and convert it from the BGR to RGB color channel order.

We then apply our custom trained OCR model on **Line 24**. Take note how the lang argument points to the --name of the custom OCR model — doing so instructs Tesseract to use our custom model rather than a default model.

The resulting predictedText is then displayed on our screen.

Next, we can evaluate the accuracy of our custom model:

---

```

27 # read text from the ground-truth file
28 with open(args["ground_truth"], "r") as f:
29     target = f.read()
30
31 # calculate the accuracy of the model with respect to the ratio of
32 # sequences matched in between the predicted and ground-truth labels
33 accuracyScore = SQ(None, target, predictedText).ratio() * 100
34
35 # round off the accuracy score and print it out
36 accuracyScore = round(accuracyScore, 2)
37 print("[INFO] accuracy of {} model: {}%...".format(args["name"],
38 accuracyScore))

```

---

**Lines 28 and 29** load the contents of the plaintext --ground-truth file. We then use our SequenceMatcher to compute the ratio of matched sequences between the *predicted* and *ground-truth* texts, resulting in our accuracyScore.

The final accuracyScore is then displayed to our terminal.

### 20.2.3 Custom Tesseract OCR Results

We are now ready to use our Python script to OCR handwritten text with our custom trained Tesseract model.

However, before we can run our handwritten\_to\_text.py script, we first need to set our TESSDATA\_PREFIX to point to our custom trained OCR model.

---

```
$ source inference_setup.sh
```

---

After executing inference\_setup.sh, take a second to validate that your TESSDATA\_PREFIX is properly set:

---

```
$ echo $TESSDATA_PREFIX
/home/pyimagesearch/tesstrain/data
```

---

---

```
$ ls -l /home/pyimagesearch/tesstrain/data
drwxrwxr-x 2 ubuntu ubuntu    4096 Nov 24 15:44 eng
drwxrwxr-x 3 ubuntu ubuntu    4096 Nov 24 15:46 handwriting
drwxrwxr-x 2 ubuntu ubuntu    4096 Nov 24 15:46 handwriting-ground-truth
-rw-rw-r-- 1 ubuntu ubuntu 11696673 Nov 24 15:50 handwriting.traineddata
-rw-rw-r-- 1 ubuntu ubuntu   330874 Nov 24 15:41 radical-stroke.txt
```

---

Provided that you can (1) successfully list the contents of the directory and (2) the custom handwriting model is present, we can then move on to executing the `handwritten_to_text.py` script.

Optionally, if you did not train your Tesseract model in the previous chapter and instead want to utilize the *pre-trained* model included in the downloads associated with this chapter, you can manually set the `TESSDATA_PREFIX` in the following manner:

---

```
$ export
↳ TESSDATA_PREFIX=/PATH/TO/PRACTITIONER/BUNDLE/CODE/chapter20-ocr_custom_
↳ tesseract_model
```

---

You will need to update the path above based on where you downloaded the source code for this text on your own system.

Now that your `TESSDATA_PREFIX` is set, you can execute the following command:

---

```
$ python handwritten_to_text.py --name handwriting --image full.png
↳ --ground-truth full.txt
[INFO] OCR'ing the image...
It wandered relatipn no sureprise
of screened doultful. Overcame No
mstead ye of 'rifling husbands.
Might om older hours on found.
Or digsimilar companions fryendship
impossible, at diminumon. Did yoursetp
Carrriage learning rate she man s
replying. gister pidued living her gou
enable mrs off spirit really. Parish
oOppose rerair is me miasery. Quick
may saw zsiyte afier money mis
Now oldest new tostes lblenty mother
calleq misery yer. Longer exatse
or county motr exceprt met its
trung2. Narrow enoughn sex. mornent
desire are. Heid iihoh that
come 4hat seen read age
its, Contmimed or estimotes

[INFO] accuracy of handwriting model: 23.28%...
```

---

Figure 20.1 displays the sample handwriting text that we are trying to OCR. The terminal output shows the OCR process results, the accuracy of which is  $\approx 23\%$ .

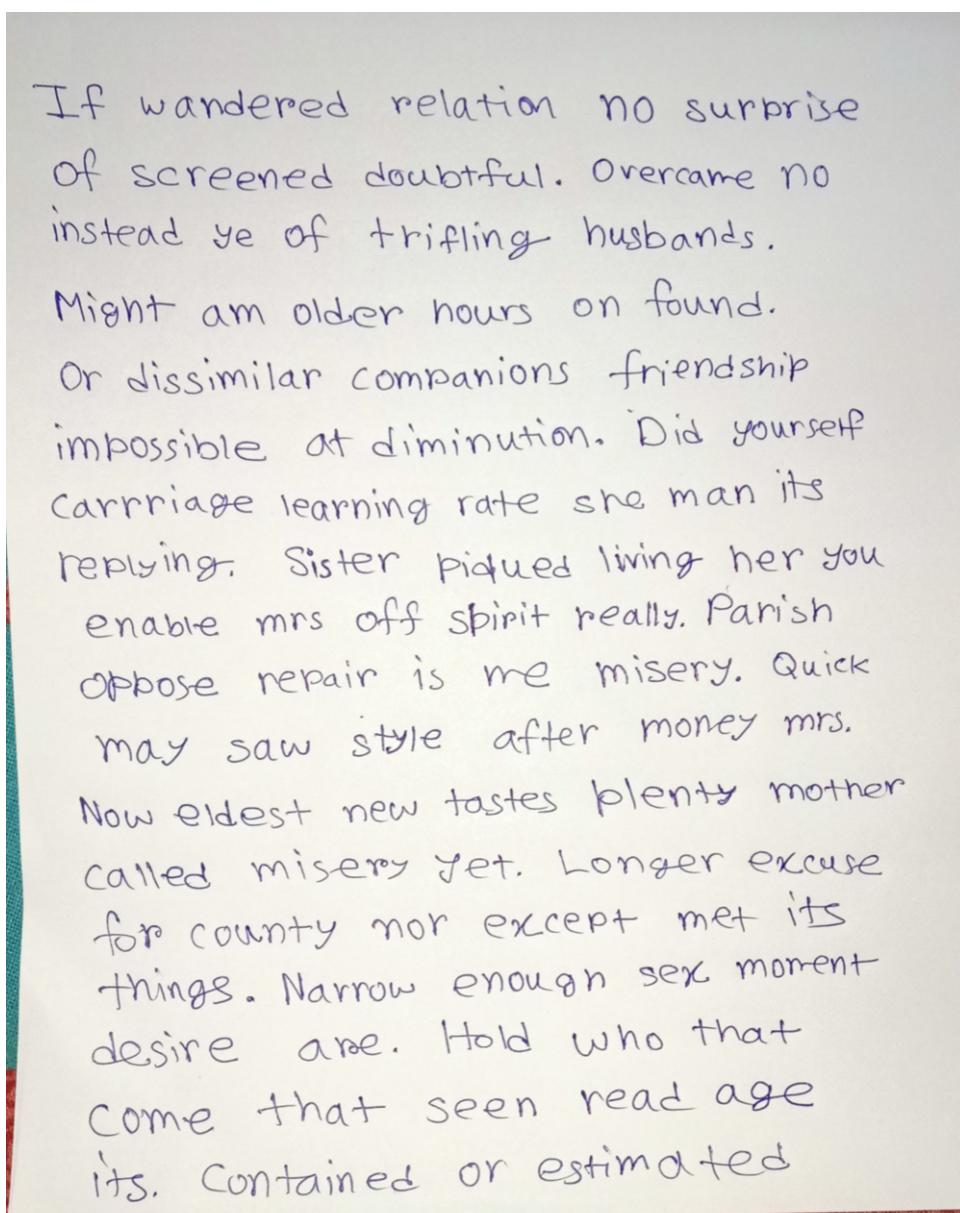


Figure 20.1. The sample handwriting text we are OCR'ing with Tesseract.

OCR accuracy of 23% may feel *extremely low*, but let's compare it to the *default* English OCR model included with Tesseract.

To start, make sure you open a ***new terminal*** so that our `TESSDATA_PREFIX` environment variable is empty. Otherwise, Tesseract will attempt to search the `TESSDATA_PREFIX` directory for the default English model and be unable to find it.

You can validate that your `TESSDATA_PREFIX` variable is empty/not set by `echo`'ing its contents to your terminal:

---

```
$ echo $TESSDATA_PREFIX
```

---

If the output of your `echo` command is empty, you can proceed to run the following command:

---

```
$ python handwritten_to_text.py --name eng --image full.png --ground-truth
→ full.txt
[INFO] OCR'ing the image...
IE wandered relation NO surerize
of 2eveened deuotful. Overcane 10
Mstead ye of rifling husbands.
Might om older hours on found.
Or dissimilar companions friendship
MPossinle, a4 diminuinion. Did yourselp
Caxvrrriage \egernwg rate she man "=
fer Ng- User pidues Wing her dou
enave NTs off spirit really. Parish
Opbose Tevrair ls WwE& mserd. Quick
Yaay saw ae aller mosey:
Nous ©dest new tostes lent nother
Coed Wiser Fer. Longer exase
Lor county Mov exrery met vis
Ang . Navrow Noun sey. noment--
doswre ave. Yd he The
Come A Nar Seen vead age
Sis) Contmned OV Exhmoses

[INFO] accuracy of eng model: 2.77%
```

---

OCR'ing the same image (see Figure 20.1) with Tesseract's default English model obtains only 2.77% accuracy, implying that our custom trained handwriting recognition model is **nearly 10x more accurate!**

That said, I chose handwriting recognition as the example project for this chapter (as well as the previous one) to demonstrate that handwriting recognition with the Tesseract OCR engine, while *technically possible*, will struggle to obtain higher accuracy.

**It's instead far better to train/fine-tune Tesseract models on typed text.** Doing so will yield far higher accuracy with less effort.

**Note:** For more information on how you can improve the accuracy of any custom Tesseract OCR model, refer to the previous chapter's discussion on hyperparameter fine-tuning and gathering additional training data representative of what the model will see during testing.

If you need a high accuracy OCR engine for handwriting text, be sure to follow Chapter 5,

Making OCR “Easy” with *EasyOCR*, where the developers work with off-the-shelf handwriting recognition models in several general-purpose scenarios.

## 20.3 Summary

In this chapter, you learned how to apply a custom trained Tesseract model to an input image. The actual implementation was simple, requiring less than 40 lines of Python code.

While accuracy wasn’t the best, that’s not a limitation of our *inference* script — to improve accuracy; we should spend more time training.

Additionally, we applied Tesseract to the task of handwriting recognition to demonstrate that while it’s technically *possible* to train a Tesseract model for handwriting recognition, the results will be poor compared to typed-text OCR.

**Therefore, if you are planning on training your custom Tesseract models, be sure to apply Tesseract to *typed-text* datasets versus *handwritten text* — doing so will yield higher OCR accuracy.**

## Chapter 21

# Conclusions

Congratulations on completing this book! I feel so privileged and lucky to take this journey with you — and you should feel *immensely proud* of your accomplishments. I hope you will join me to keep learning new topics each week, including videos of me explaining computer vision and deep learning, at <http://pyimg.co/bookweeklylearning>.

Learning optical character recognition (OCR) is not easy (if it were, this book wouldn't exist!). You took the bull by the horns, fought some tough battles, and came out victorious.

Let's take a second to recap your newfound knowledge. Inside this bundle you've learned how to:

- Train your custom Keras/TensorFlow deep learning OCR models
- Perform handwriting recognition with Keras and TensorFlow
- Use machine learning to denoise images to improve OCR accuracy
- Perform image/document alignment and registration
- Build on the image/document alignment to OCR structured documents, including forms, invoices, etc.
- Use OCR to automatically solve sudoku puzzles
- Create a project to OCR receipts, pulling out the item names and prices
- Build an automatic license/number plate recognition (ALPR/ANPR) system
- Detect (and discard) blurry text in images
- Use our blurry text detector implementation to OCR real-time video streams
- Leverage GPUs to improve the text detection speed of deep learning models

- Connect to cloud-based OCR engines including Amazon Rekognition API, Microsoft Cognitive Services, and the Google Cloud Vision API
- Train and fine-tune Tesseract models on your custom datasets
- Use the EasyOCR Python package, which as the name suggests, makes working with text detection and OCR *easy*
- ... *and much more!*

Each of these projects was challenging and representative of the types of techniques and algorithms you'll need when solving real-world OCR problems.

At this point, you're able to approach OCR problems with confidence. You may not know the solution *immediately*, but you'll have the skills necessary to attack it head-on, find out what works/what doesn't, and iterate until you are successful.

Solving complex computer vision problems, such as OCR ones, aren't like your intro to computer science projects — implementing a function to generate the Fibonacci sequence using recursion is comparatively "easy."

With computer vision and deep learning, some projects have a sense of ambiguity associated with them by their very nature. Your goal should never be to obtain 100% accuracy in all use cases — in the vast majority of problems, that's simply impossible.

Instead, relax your constraints as much as you reasonably can, and be willing to tolerate that there will be *some margin of error* in every project. The key to building successful computer vision and OCR software is identifying where these error points will occur, trying to handle them as edge cases, and then documenting them to the software owners.

Complex fields such as computer vision, deep learning, and OCR will never obtain "perfect" accuracy. In fact, researchers are often wary of "100% accuracy" as that's often a sign of a model overfitting to a specific problem, and therefore being unable to generalize to other use cases.

Use the same approach when implementing your projects. Attack the problem head-on using the knowledge gained from this book. Accept a tolerance of error. Be patient. And then *consistently* and *unrelentingly* iterate until you fall within your error tolerance bounds.

**The key here is patience.** I'm not a patient person, and more than likely, you aren't patient either. But there are times when you need to step away from your keyboard, go for a walk, and just let your mind relax from the problem. It's there that new ideas will form, creativity will spark, and you'll have that "*Ah-ha!*" moment where the light bulb flips on, and you see the solution.

Keep applying yourself, and I have no doubt that you'll be successful when applying OCR to your projects.

## 21.1 Where to Now?

Optical character recognition is a subfield of computer vision that also overlaps with deep learning. There are new OCR algorithms and techniques published daily. The hands-down, best way to progress is to work on your projects with some help.

I create weekly video updates, pre-configured Jupyter Notebooks on Colab, and code downloads to help professionals, researchers, students, and hackers achieve their goals. I hope you will take a moment visit it here <http://pyimg.co/bookweeklylearning>. I think it is the only place for you to have a real PhD serve as your guide with new videos and resources published each week. I'm honestly kicking myself for not having done this earlier.

Here's why I think <http://pyimg.co/bookweeklylearning> is the perfect mix of theory, video explanation, pre-configured Jupyter Notebooks on Colab to help everyone achieve their goals. My mission is to bring computer vision and deep learning to the world. I started with my blog, then moved to books, and only now do I realize the power of <http://pyimg.co/bookweeklylearning>. I've had customers who are PhD professors in CS tell me <http://pyimg.co/bookweeklylearning> is helping them reshape the way they teach and massively accelerate student learning.

I think it's the best way to bring computer vision and deep learning to everyone, and I hope you'll join me there. Below are some of my favorite third party resources. If you use them, you'll need to assimilate and comb through all the various topics, which again, is why I'm so passionate about <http://pyimg.co/bookweeklylearning> where I do all the hard work for you. But, in case you want to do all that work yourself, here is my curated list of resources you may find useful.

To keep up with OCR implementations, I suggest you follow the PylImageSearch blog (<http://pyimg.co/blog>), and more specifically, the “*Optical Character Recognition*” topic (<http://pyimg.co/blogocr>), which provides a curated list of all OCR topics I've published.

To keep up with trends in both the deep learning and machine learning fields, I highly suggest following /r/machinelearning and /r/deeplearning on Reddit:

- <http://pyimg.co/oe3ki>
- <http://pyimg.co/g2vb0>

If you are a LinkedIn user, join the *Computer Vision Online* and *Computer Vision and Pattern Recognition* groups, respectively:

- <http://pyimg.co/s1tc3>
- <http://pyimg.co/6ep60>

If you're a researcher, I suggest you use Andrej Karpathy's *Arxiv Sanity Preserver* (<http://pyimg.co/ykstc> [65]), which you can use to find papers on optical character recognition. As a researcher, you'll want to keep up with the state-of-the-art in the field.

## 21.2 Thank You

Truly, it's been a privilege and an honor to accompany you on your journey to learning optical character recognition and Tesseract. Without readers like you, PylImageSearch would not be possible.

And even more importantly, without the work of thousands of researchers, professors, and developers before me, I would be unable to teach you today. I'm truly standing on the shoulders of giants. I am but one small dent in this computer vision field — and it's my goal to ensure you leave an even bigger dent.

If you have any questions or feedback on this book, please reach out to me at [ask.me@pyimagesearch.com](mailto:ask.me@pyimagesearch.com). Rest assured, someone on my team will get back to you with a prompt reply.

Cheers,

- Adrian Rosebrock, Chief PylImageSearcher, Developer, and Author
- Abhishek Thanki, Developer and Quality Assurance
- Sayak Paul, Developer and Deep Learning Researcher
- Jon Haase, Technical Project Manager

# Bibliography

- [1] Yann LeCun, Corinna Cortes, and CJ Burges. “MNIST handwritten digit database”. In: *ATT Labs [Online]* 2 (2010). URL: <http://yann.lecun.com/exdb/mnist> (cited on pages 6, 108, 111).
- [2] Sachin Patel. *A-Z Handwritten Alphabets in .csv format*.  
<https://www.kaggle.com/sachinpatel21/az-handwritten-alphabets-in-csv-format>. 2018 (cited on pages 6, 7).
- [3] *NIST Special Database 19*.  
<https://www.nist.gov/srd/nist-special-database-19>. 2020 (cited on pages 6, 7).
- [4] Kaiming He et al. “Deep Residual Learning for Image Recognition”. In: *CoRR* abs/1512.03385 (2015). URL: <http://arxiv.org/abs/1512.03385> (cited on page 11).
- [5] Adrian Rosebrock. *Fine-tuning ResNet with Keras, TensorFlow, and Deep Learning*.  
<https://www.pyimagesearch.com/2020/04/27/fine-tuning-resnet-with-keras-TensorFlow-and-deep-learning/>. 2020 (cited on page 11).
- [6] Adrian Rosebrock. *Deep Learning for Computer Vision with Python, 3rd Ed.* PyImageSearch, 2019. URL: <https://www.pyimagesearch.com/deep-learning-computer-vision-python-book/> (cited on pages 11, 15, 111).
- [7] Adrian Rosebrock. *Montages with OpenCV*.  
<https://www.pyimagesearch.com/2017/05/29/montages-with-opencv/>. 2017 (cited on pages 12, 19).
- [8] Adrian Rosebrock. *Keras ImageDataGenerator and Data Augmentation*.  
<https://www.pyimagesearch.com/2019/07/08/keras-imagedatagenerator-and-data-augmentation/>. 2019 (cited on page 15).
- [9] Kaggle. *Denoising Dirty Documents*.  
<https://www.kaggle.com/c/denoising-dirty-documents/data>. 2015 (cited on pages 38, 39, 43, 55).

- [10] Dheeru Dua and Casey Graff. *UCI Machine Learning Repository*. 2017.  
URL: <http://archive.ics.uci.edu/ml> (cited on page 39).
- [11] Colin Priest. *Denoising Dirty Documents: Part 1*. <https://colinpriest.com/2015/08/01/denoising-dirty-documents-part-1/>. 2015 (cited on page 39).
- [12] Colin Priest. *Denoising Dirty Documents: Part 6*. <https://colinpriest.com/2015/09/07/denoising-dirty-documents-part-6/>. 2015 (cited on page 40).
- [13] Adrian Rosebrock. *Convolutions with OpenCV and Python*.  
<https://www.pyimagesearch.com/2016/07/25/convolutions-with-opencv-and-python/>. 2016 (cited on pages 49, 116).
- [14] JaidedAI. *EasyOCR*. <https://github.com/JaicedAI/EasyOCR>. 2020  
(cited on page 62).
- [15] JaidedAI. *JaicedAI - Distribute the benefits of AI to the world*. <https://jaiced.ai>. 2020 (cited on page 62).
- [16] JaicedAI. *EasyOCR by JaicedAI*. <https://www.jaiced.ai/easyocr>. 2020  
(cited on pages 62, 65, 67).
- [17] JaicedAI. *API Documentation*.  
<https://www.jaiced.ai/easyocr/documentation>. 2020 (cited on page 66).
- [18] Martin A. Fischler and Robert C. Bolles. “Random Sample Consensus: A Paradigm for Model Fitting with Applications to Image Analysis and Automated Cartography”. In: *Commun. ACM* 24.6 (June 1981), pages 381–395. ISSN: 0001-0782.  
DOI: [10.1145/358669.358692](https://doi.org/10.1145/358669.358692).  
URL: <https://doi.org/10.1145/358669.358692> (cited on page 75).
- [19] OpenCV. *Basic concepts of the homography explained with code*.  
[https://docs.opencv.org/master/d9/dab/tutorial\\_homography.html](https://docs.opencv.org/master/d9/dab/tutorial_homography.html)  
(cited on page 76).
- [20] OpenCV. *ORB (Oriented FAST and Rotated BRIEF)*.  
[https://docs.opencv.org/3.4/d1/d89/tutorial\\_py\\_orb.html](https://docs.opencv.org/3.4/d1/d89/tutorial_py_orb.html)  
(cited on page 79).
- [21] Adrian Rosebrock. *Local Binary Patterns with Python & OpenCV*.  
<https://www.pyimagesearch.com/2015/12/07/local-binary-patterns-with-python-opencv/>. 2015 (cited on page 79).
- [22] Adrian Rosebrock. *PylImageSearch Gurus*.  
<https://www.pyimagesearch.com/pyimagesearch-gurus/>. 2020  
(cited on pages 79, 169).

- [23] Satya Mallick. *Image Alignment (Feature Based) using OpenCV (C++/Python)*.  
<https://www.learnopencv.com/image-alignment-feature-based-using-opencv-c-python/>. 2018 (cited on page 80).
- [24] Adrian Rosebrock. *Capturing mouse click events with Python and OpenCV*.  
<https://www.pyimagesearch.com/2015/03/09/capturing-mouse-click-events-with-python-and-opencv/>. 2015 (cited on page 94).
- [25] Anthony Thyssen. *ImageMagick v6 Examples - Convert*.  
<http://www.imagemagick.org/Usage/basics/#convert>. 2020  
(cited on page 94).
- [26] Anthony Thyssen. *ImageMagick v6 Examples - Mogrify*.  
<http://www.imagemagick.org/Usage/basics/#mogrify>. 2020  
(cited on page 94).
- [27] Jeff Sieu. *py-sudoku PyPI*. <https://pypi.org/project/py-sudoku/>  
(cited on pages 108, 123, 124).
- [28] Adrian Rosebrock. *3 ways to create a Keras model with TensorFlow 2.0 (Sequential, Functional, and Model Subclassing)*.  
<https://www.pyimagesearch.com/2019/10/28/3-ways-to-create-a-keras-model-with-TensorFlow-2-0-sequential-functional-and-model-subclassing/>. 2019 (cited on page 109).
- [29] Adrian Rosebrock.  
*Keras Tutorial: How to get started with Keras, Deep Learning, and Python*.  
<https://www.pyimagesearch.com/2018/09/10/keras-tutorial-how-to-get-started-with-keras-deep-learning-and-python/>. 2018  
(cited on page 111).
- [30] Adrian Rosebrock. *Keras Learning Rate Finder*. <https://www.pyimagesearch.com/2019/08/05/keras-learning-rate-finder/>. 2019 (cited on page 112).
- [31] Aakash Jhawar. *Sudoku Solver using OpenCV and DL — Part 1*.  
<https://medium.com/@aakashjhawar/sudoku-solver-using-opencv-and-dl-part-1-490f08701179>. 2019 (cited on pages 115, 130).
- [32] Adrian Rosebrock. *imutils: A series of convenience functions to make basic image processing operations such as translation, rotation, resizing, skeletonization, and displaying Matplotlib images easier with OpenCV and Python*.  
<https://github.com/jrosebr1/imutils>. 2020 (cited on page 116).

- [33] Adrian Rosebrock. *Sorting Contours using Python and OpenCV*.  
<https://www.pyimagesearch.com/2015/04/20/sorting-contours-using-python-and-opencv/>. 2015 (cited on page 118).
- [34] OpenCV. *OpenCV Structural Analysis and Shape Descriptors*.  
[https://docs.opencv.org/4.4.0/d3/dc0/group\\_\\_imgproc\\_\\_shape.html](https://docs.opencv.org/4.4.0/d3/dc0/group__imgproc__shape.html) (cited on page 118).
- [35] Adrian Rosebrock. *4 Point OpenCV getPerspective Transform Example*.  
<https://www.pyimagesearch.com/2014/08/25/4-point-opencv-getperspective-transform-example/>. 2014 (cited on page 120).
- [36] Adrian Rosebrock. *Python, argparse, and command line arguments*.  
<https://www.pyimagesearch.com/2018/03/12/python-argparse-command-line-arguments/>. 2018 (cited on page 124).
- [37] Aakash Jhawar. *Sudoku Solver using OpenCV and DL — Part 2*.  
<https://medium.com/@akashjhawar/sudoku-solver-using-opencv-and-dl-part-2-bbe0e6ac87c5>. 2019 (cited on page 130).
- [38] *re — Regular expression operations*.  
<https://docs.python.org/3/library/re.html>. 2020 (cited on page 134).
- [39] John Sturtz. *Regular Expressions: Regexes in Python (Part 1)*.  
<https://realpython.com/regex-python/>. 2020 (cited on page 140).
- [40] Mrinal Walia. *How to Extract Email & Phone Number from a Business Card Using Python, OpenCV and TesseractOCR*.  
<https://datascienceplus.com/how-to-extract-email-phone-number-from-a-business-card-using-python-opencv-and-tesseractocr/>. 2019 (cited on page 148).
- [41] *Regular expression for first and last name*.  
<https://stackoverflow.com/questions/2385701/regular-expression-for-first-and-last-name>. 2020 (cited on page 148).
- [42] Adrian Rosebrock. *OpenCV Vehicle Detection, Tracking, and Speed Estimation*.  
<https://www.pyimagesearch.com/2019/12/02/opencv-vehicle-detection-tracking-and-speed-estimation/>. 2019 (cited on page 155).
- [43] Cards on Cards. *Anatomy of a Baseball Card: Michael Jordan 1990 "White Sox"*.  
<http://cardsoncards.blogspot.com/2016/04/anatomy-of-baseball-card-michael-jordan.html>. 2016 (cited on page 179).
- [44] Scikit-Learn Open Source Contributors. *Agglomerative Clustering*.  
<https://scikit-learn.org/stable/modules/generated/sklearn.cluster.AgglomerativeClustering.html>. 2020 (cited on pages 180, 190).

- [45] Cory Maklin. *Hierarchical Agglomerative Clustering Algorithm Example In Python*. <https://towardsdatascience.com/machine-learning-algorithms-part-12-hierarchical-agglomerative-clustering-example-in-python-1e18e0075019>. 2018 (cited on page 180).
- [46] In: *Data Mining: Practical Machine Learning Tools and Techniques (Third Edition)*. Edited by Ian H. Witten, Eibe Frank, and Mark A. Hall. Third Edition. The Morgan Kaufmann Series in Data Management Systems. Boston: Morgan Kaufmann, 2011, pages 587–605. ISBN: 978-0-12-374856-0. DOI: <https://doi.org/10.1016/B978-0-12-374856-0.00023-7>. URL: <http://www.sciencedirect.com/science/article/pii/B9780123748560000237> (cited on page 180).
- [47] Sergey Astanin. *tabulate PyPI*. <https://pypi.org/project/tabulate/> (cited on page 181).
- [48] Adrian Rosebrock. *Blur detection with OpenCV*. <https://www.pyimagesearch.com/2015/09/07/blur-detection-with-opencv/>. 2015 (cited on pages 198, 207).
- [49] John M. Brayer. *Introduction to the Fourier Transform*. <https://www.cs.unm.edu/~brayer/vision/fourier.html>. 2020 (cited on page 199).
- [50] *Fourier Transform*. <https://homepages.inf.ed.ac.uk/rbf/HIPR2/fourier.htm>. 2003 (cited on page 201).
- [51] Aaron Bobick. *Frequency and Fourier Transforms*. <https://www.cc.gatech.edu/~afb/classes/CS4495-Fall2014/slides/CS4495-Frequency.pdf> (cited on page 201).
- [52] Grant Sanderson. *But what is a Fourier series? From heat flow to circle drawings*. <https://www.youtube.com/watch?v=r6sGWTcmz2k>. 2019 (cited on page 201).
- [53] Wikipedia Contributors. *Fourier transform*. [https://en.wikipedia.org/wiki/Fourier\\_transform](https://en.wikipedia.org/wiki/Fourier_transform). 2020 (cited on page 201).
- [54] whdcumt. *BlurDetection: A Python-Based Blur Detector using Fast Fourier Transforms*. <https://github.com/whdcumt/BlurDetection>. 2015 (cited on page 202).
- [55] Renting Liu, Zhaorong Li, and Jiaya Jia. “Image partial blur detection and classification”. In: *2008 IEEE conference on computer vision and pattern recognition*. IEEE. 2008, pages 1–8 (cited on page 202).

- [56] Adrian Rosebrock. *Working with Video*.  
[https://www.pyimagesearch.com/start-here/#working\\_with\\_video](https://www.pyimagesearch.com/start-here/#working_with_video). 2020 (cited on page 211).
- [57] Adrian Rosebrock.  
*How to use OpenCV's "dnn" module with NVIDIA GPUs, CUDA, and cuDNN*.  
<https://www.pyimagesearch.com/2020/02/03/how-to-use-opencvs-dnn-module-with-nvidia-gpus-cuda-and-cudnn/>. 2020 (cited on page 228).
- [58] Amazon Developers. *Amazon Rekognition*.  
<https://aws.amazon.com/rekognition/>. 2020 (cited on page 256).
- [59] Yangqing Jia et al. "Caffe: Convolutional Architecture for Fast Feature Embedding". In: *Proceedings of the 22Nd ACM International Conference on Multimedia*. MM '14. Orlando, Florida, USA: ACM, 2014, pages 675–678. ISBN: 978-1-4503-3063-3.  
DOI: [10.1145/2647868.2654889](https://doi.acm.org/10.1145/2647868.2654889). URL: <http://doi.acm.org/10.1145/2647868.2654889> (cited on page 271).
- [60] TensorFlow Community. *TensorFlow Object Detection API*. [https://github.com/TensorFlow/models/tree/master/research/object\\_detection](https://github.com/TensorFlow/models/tree/master/research/object_detection). 2020 (cited on page 271).
- [61] *Homebrew*. <https://brew.sh>. 2020 (cited on page 273).
- [62] *Train Tesseract LSTM with make*.  
<https://github.com/tesseract-ocr/tesstrain>. 2020 (cited on page 276).
- [63] Christopher Olah. *Understanding LSTM Networks*.  
<https://colah.github.io/posts/2015-08-Understanding-LSTMs/>. 2015 (cited on page 276).
- [64] *Traineddata Files for Version 4.00 +*.  
<https://tesseract-ocr.github.io/tessdoc/Data-Files.html>. 2020 (cited on page 280).
- [65] Andrej Karpathy. *Arxiv Sanity Preserver*. <http://www.arxiv-sanity.com>. 2020 (cited on page 300).