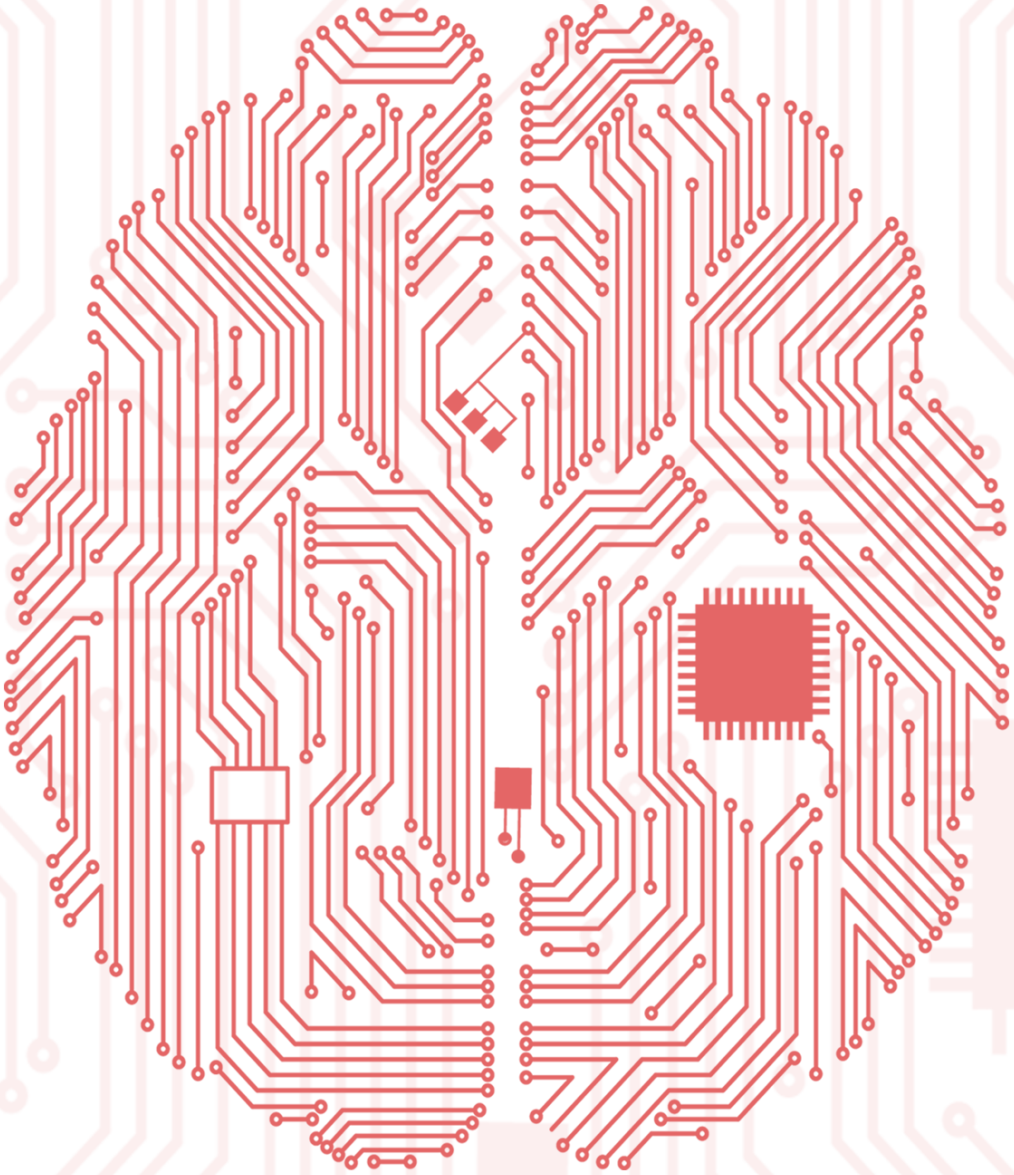
DEEP LEARNING FOR COMPUTER VISION



WITH PYTHON

Dr. Adrian Rosebrock

pyimagesearch

Deep Learning for Computer Vision with Python

Bonus Bundle

Dr. Adrian Rosebrock

3rd Edition (3.0)

PUBLISHED BY PYIMAGESEARCH

PYIMAGESEARCH.COM

Third printing, November 2019

To my father, Joe; my wife, Trisha;
and the family beagles, Josie and Jemma.
Without their constant love and support,
this book would not be possible.

Contents

Companion Website

Thank you for picking up a copy of *Deep Learning for Computer Vision with Python!* To accompany this book I have created a companion website which includes:

- **Up-to-date installation instructions** on how to configure your development environment
- Instructions on how to use the **pre-configured Ubuntu VirtualBox virtual machine** and **Amazon Machine Image (AMI)**
- **Supplementary material** that I could not fit inside this book
- **Frequently Asked Questions (FAQs)** and their suggested fixes and solutions

Additionally, you can use the "Issues" feature inside the companion website to report any bugs, typos, or problems you encounter when working through the book. I don't expect many problems; however, this is a brand new book so myself and other readers would appreciate reporting any issues you run into. From there, I can keep the book updated and bug free.

To create your companion website account, just use this link:

http://pyimg.co/fnkxk

Take a second to create your account now so you'll have access to the supplementary materials as you work through the book.

1. Object Detection with RetinaNet

In this chapter you'll learn how to use Keras and the RetinaNet object detection framework to detect and localize logos of popular brands in images. I really enjoyed working with RetinaNet; however, there are a few reasons that prevented me from including it in the formal release of this book.

The first is that RetinaNet, while an active open source project [1], does not have the support that TensorFlow Object Detection (TFOD) API [2] has. There are some very active developers working on RetinaNet, but when writing a book, one must ensure that content authored in a technical field should be relevant for *at least* a few years.

It could be the case that the RetinaNet project will still be active a few years from now — and it could also be the case that the code still executes without a problem. But if we had to take bets, the TFOD API, with the large support from Google and other developers, will likely still be around. Thus, the TFOD API made a better teaching tool.

Additionally, I ran into a few issues when using the RetinaNet implementation, mostly involving *validation* during training and *evaluation* after training had completed — neither method was working properly, making it near impossible to determine if the network was overfitting outside of exhaustive visual validation.

This problem could be my fault (user error) or it could be a bug in the actual library and its associated tools. I worked with the package for a bit but I could not determine the root cause. And if I struggled with it, I knew others would struggle with it as well.

That said, RetinaNet is a good package and I believe the authors and contributors deserve a *huge* round of applause. Implementing state-of-the-art deep learning object detectors *by hand* is not an easy task — they pulled it off and they deserve all the credit.

In the rest of this chapter you'll learn what RetinaNet is and how to use RetinaNet to create your own custom object detectors.

1.1 What is RetinaNet?

RetinaNet was first introduced by Lin et al. in their 2017 paper, *Focal Loss for Dense Object Detection* [3]. RetinaNet can be considered a cousin to *both* the R-CNN family (Girshick is actually the third author on the RetinaNet paper) and the SSD family of object detectors.

In their paper, Lin et al. study what makes two-stage detectors, such as R-CNNs, (usually) more accurate than one-stage detectors such as SSD and YOLO. In a two stage detector, we apply our classifier to a sparse set of candidate object locations; however,in a one-stage detector, we instead apply our classifier over a dense, regularly sampled set of possible object locations [3].

Two-stage detectors can sometimes be more accurate, but at the expense of computational complexity. One-stage detectors tend to be significantly faster, simpler, and more intuitive, but may not be as accurate.

Through their studies, Lin et al. discovered the fundamental issue with one-stage detectors is the "extreme foreground-background class imbalance" (typically we have far more background examples than foreground, object examples).

To remedy this, Lin et al. introduce a novel extension to standard cross-entropy loss called *Focal Loss*. The Focal Loss function down-weights the loss assigned to confident (according to the model) detected examples.

Therefore, instead of allowing the network to be overwhelmed by "easy" hard-negatives, the Focal Loss instead "forces" the network to learn from the more challenging hard-negatives, thereby increasing object detection accuracy.

In their experiments, Lin et al. define a dense, one-stage detector called *RetinaNet* which uses the Focal Loss function. ResNet is used as the default base network which we will also be using in this chapter as well.

Their results demonstrated that RetinaNet could match the speed of existing one-stage detectors (SSD and YOLO) while obtaining superior object detection accuracy than two-stage, R-CNN inspired detectors. RetinaNet therefore seeks to obtain the best of both worlds: speed and accuracy.

1.2 Installing RetinaNet

The RetinaNet implementation we will be using can be found on their GitHub project page [1]. This implementation is under active development. Since install instructions can and will change, I decided to move the development environment configuration instructions to the companion website.

You can find the Keras + RetinaNet install instructions on this page: http://pyimg.co/3j90k

If you do not already have an account on the companion website, please refer to the first few pages of this book where a link is included to register for the supplementary material.

1.3 The Logo Dataset

The dataset we are going to use today is ROMYNY Logo 2016 [4], a collection of twenty popular logo classes, such as Adidas, Facebook, BMW, Coca Cola, and others.

The logos dataset was put together by Frank Fotso, a PyImageSearch reader and PyImageSearch Gurus member (http://pyimg.co/gurus) who is doing his research on computer vision, deep learning, and object detection. The dataset released by Fotso is actually a *subset* of the data he is working with for his research. Fotso expects to release an *even larger dataset* as he wraps up his research.

There are a total of 935 images in the dataset, with anywhere between 19-159 images per class, making the dataset unbalanced, but representative of what we may encounter in the real-world. Table 1.1 includes the full breakdown on number of images per class.

1.3.1 How Do I Download the Logos Dataset?

I have included the logos dataset in the download associated with this bonus chapter so there is no need to search for it online. I decided to distribute the dataset along with this guide as I needed to:

1. Modify the original training testing split to allocate more data to training (especially since the testing and evaluation scripts were not working properly)

Logo	**Image Count**
Adidas-Pict	138
Adidas-Text	121
Aldi	122
Allianz-Pict	132
Allianz-Text	139
Amazon	81
Apple	67
Atletico_Madrid	35
Audi-Pict	79
Audi-Text	19
BWM	84
Burger_king	19
BMW	84
CocaCola	158
FC_Barcelona	65
FC_Bayern_Munchen	78
Facebook-Pict	11
Facebook-text	72
Ferrari-Pict	62
Ferrari-Text	48
eBay	115

Table 1.1: A table summary of the the number of classes per logo in the logos dataset. There exist heavy class imbalance between some of the logos.

2. Update the directory structure to make it easier to parse
3. Address a problem where each object in an image was potentially annotated multiple times

Again, you can find the logos dataset in your download associated with this bonus chapter.

1.3.2 Project Structure

Before we train our RetinaNet object detector, let's first review the project structure:

```
|--- retinanet
|    |--- config
|    |    |--- __init__.py
|    |    |--- logos_config.py
|    |--- logos
|    |    |--- annotations
|    |    |--- images
|    |    |--- train.txt
|    |    |--- test.txt
|    |    |--- resnet50_coco_best_v2.1.0.h5
|    |--- build_logos.py
|    |--- predict.py
```

Just as we have done many times in this book, we are going to create a `config` module to store any necessary configurations to create the dataset and train the model. These configurations will be stored inside `logos_config.py`.

The `logos` directory contains the logos dataset itself. Here you'll find the `annotations` (bounding box information), `images`, and training and testing split files. Take the time now and spend a few minutes to explore the dataset using your terminal.

In particular, I would recommend that you:

1. Investigate the `logos` dataset using the `ls` command to see how it is structured.
2. Open up the `train.txt` and `test.txt` file and note how each line contains a single integer — the unique ID of each image in the dataset.
3. List the contents of the `images` directory and see how each image filename has the format `{IMAGE_ID}.jpg` — this filename structure allows us to map the unique image ID to a filename.
4. Similarly, you should explore the `annotations` directory and open up a few of the XML files to see how they are structured.

We will learn how to parse these annotation files and build our dataset in Section 1.3.4. The `resnet50_coco_best_v2.1.0.h5` file is a Keras + RetinaNet model that has been pre-trained on the COCO dataset [5]. We will be fine-tuning this model during training. Go ahead and copy your local copy of `resnet50_coco_best_v2.1.0.h5` into your `logos` directory now.

After training our network (using the `retinanet-train` command), we can use the `predict.py` script to test our object detector on new images.

1.3.3 The Configuration File

Our configuration file for the logos dataset is straightforward. Open up `logos_config.py` and take a look:

```
1 # import the necessary packages
2 import os
3
4 # initialize the base path for the logos dataset
```

```
5  BASE_PATH = "logos"
6
7  # build the path to the annotations and input images
8  ANNOT_PATH = os.path.sep.join([BASE_PATH, "annotations"])
9  IMAGES_PATH = os.path.sep.join([BASE_PATH, "images"])
10
11 # build the path to the *input* training and testing .txt files
12 TRAIN_TXT = os.path.sep.join([BASE_PATH, "train.txt"])
13 TEST_TXT = os.path.sep.join([BASE_PATH, "test.txt"])
14
15 # build the path to the *output* training and test .csv files
16 TRAIN_CSV = os.path.sep.join([BASE_PATH, "retinanet_train.csv"])
17 TEST_CSV = os.path.sep.join([BASE_PATH, "retinanet_test.csv"])
18
19 # build the path to the output classes CSV file
20 CLASSES_CSV = os.path.sep.join([BASE_PATH, "retinanet_classes.csv"])
```

Line 5 defines the `BASE_PATH` to the `logos` directory. If you are using the dataset and project structure included in the download of this bonus chapter, you should not have to change the `BASE_PATH`.

Lines 8 and 9 define the paths to the input `annotations` and `images` directories, respectively.

The `TRAIN_TXT` and `TEST_TXT` file paths (**Lines 12 and 13**) refer to the training and testing splits included in the logos dataset. As we'll see in Section 1.3.4 below, we'll need to write the training and testing annotations to disk in a CSV file — **Lines 16 and 17** define the paths to these output CSV files.

Finally, **Line 20** defines the path to the class labels CSV file.

1.3.4 Building the Dataset

The Keras RetinaNet library utilizes two CSV files when training a RetinaNet object detector:

1. One CSV file contains the image path and bounding box annotation (one bounding box per line)
2. Another CSV file for the class name to integer ID mapping

Again, the annotations file should contain *one annotation per line*. If your image has multiple bounding boxes, you should use one row per bounding boxes. Below you can find an example of a bounding box annotation:

```
logos/images/452893.jpg,334,177,1035,716,CocaCola
```

The first entry is the path to the image. We then supply the bounding box coordinates in the following order:

- Start x
- Start y
- End x
- End y

Finally, we provide the human readable class label.

Let's go ahead and build our logos dataset. Open up `build_logos.py` and insert the following code:

```
1  # import the necessary packages
2  from config import logos_config as config
```

```
3  from bs4 import BeautifulSoup
4  import os
5
6  # initialize the set of classes we have encountered so far
7  CLASSES = set()
8
9  # create the list of datasets to build
10 datasets = [
11      ("train", config.TRAIN_TXT, config.TRAIN_CSV),
12      ("test", config.TEST_TXT, config.TEST_CSV),
13 ]
```

On **Lines 2-4** we import our required Python packages. **Line 2** imports our `logos_config` so we have access to it in this script.

Notice how we are once again using `BeautifulSoup` to help us parse XML files — this is because the annotations for the logos dataset are stored in PASCAL VOC format [6] which serializes annotations to disk as an XML file. Luckily, this XML file is easily to parse using the BeautifulSoup library.

Line 7 initializes the set of `CLASSES` in our dataset. We will add to this `set` as we parse the XML annotations file.

Lines 10-13 then define the lists of the `datasets` themselves, including:

1. The type of dataset.
2. The path to the input split of the data.
3. The path to the output CSV file containing the image paths and annotations.

Let's loop over each of the datasets:

```
15 # loop over the datasets
16 for (dType, inputTxt, outputCSV) in datasets:
17      # load the contents of the input data split
18      print("[INFO] starting '{}' set...".format(dType))
19      imageIDs = open(inputTxt).read().strip().split("\n")
20      print("[INFO] {} total images in '{}' set".format(
21           len(imageIDs), dType))
22
23      # open the output CSV file
24      csv = open(outputCSV, "w")
```

Line 19 loads the `inputTxt` file which contains the image IDs for the particular data split (either "training" or "testing"). **Line 24** opens our output CSV file for writing.

The next step is to loop over each of the `imageIDs` individually:

```
26      # loop over the image IDs
27      for imageID in imageIDs:
28           # build the path to the image path and annotation file
29           imagePath = "{}.jpg".format(os.path.sep.join([
30                config.IMAGES_PATH, imageID]))
31           annotPath = "{}.xml".format(os.path.sep.join([
32                config.ANNOT_PATH, imageID]))
33
34           # load the annotation file, build the soup, and initialize
35           # the set of coordinates for this particular image
36           contents = open(annotPath).read()
```

```
37          soup = BeautifulSoup(contents, "html.parser")
38          coords = set()
39
40          # extract the image dimensions
41          w = int(soup.find("width").string)
42          h = int(soup.find("height").string)
```

For each `imageID`, we need to build the path to the input image file (**Lines 29 and 30**) followed by the path to the associated XML annotation file (**Lines 31 and 32**).

Given the path to the annotation file we can load it and build our XML parser object (**Lines 36 and 37**). We also initialize `coords`, the set of extracted bounding box (x,y)-coordinates for this particular image. **Lines 41 and 42** extract the spatial dimensions (width and height) from the annotation file.

Since any given image can have *multiple* bounding boxes associated with it, we need to loop over all `object` elements in the annotation file:

```
44          # loop over all `object` elements
45          for o in soup.find_all("object"):
46              # extract the label and bounding box coordinates
47              label = o.find("name").string
48              xMin = int(o.find("xmin").string)
49              yMin = int(o.find("ymin").string)
50              xMax = int(o.find("xmax").string)
51              yMax = int(o.find("ymax").string)
52
53              # truncate any bounding box coordinates that may fall
54              # outside the boundaries of the image
55              xMin = max(0, xMin)
56              yMin = max(0, yMin)
57              xMax = min(w, xMax)
58              yMax = min(h, yMax)
```

Line 47 extracts the human readable class label from the document while **Lines 48-51** extracts the bounding box coordinates. **Lines 55-58** ensures that all bounding box coordinates do not fall *outside* the spatial dimensions of the image, an important requirement when using the Keras RetinaNet library.

When working with this dataset I noticed a problem where there were *multiple* annotations for the *same* object — in essence, each object was labeled one or more times with identical bounding box coordinates.

I'm not sure how or even why this happened, but in order to ensure there is *exactly one* bounding box for each object we need ensure each set of coordinates is represented once and only once in our CSV files:

```
60              # build a (hashable) tuple from the coordinates
61              coord = (xMin, yMin, xMax, yMax)
62
63              # if the coordinates already exist in our `coords` set,
64              # ignore the annotation (this is a peculiarity of the
65              # logos dataset)
66              if coord in coords:
67                  continue
```

We can then write the image path, bounding box coordinates, and label to the output CSV file:

```
69                # write the image path, bounding box coordinates, and
70                # label to the output CSV file
71                row = [os.path.abspath(imagePath), str(xMin), str(yMin),
72                     str(xMax), str(yMax), label]
73                csv.write("{}\n".format(",".join(row)))
74
75                # add the bounding box coordinates to our set and update
76                # the set of class labels
77                coords.add(coord)
78                CLASSES.add(label)
79
80      # close the output CSV file
81      csv.close()
```

Line 77 updates our list of `coords` to ensure we do not accidecanlly write this particular bounding box to disk multiple times. **Line 78** updates our `CLASSES` set with the `label`.

The final step is to write our `CLASSES` to disk as a CSV file:

```
83  # write the classes to file
84  print("[INFO] writing classes...")
85  csv = open(config.CLASSES_CSV, "w")
86  rows = [",".join([c, str(i + 1)]) for (i, c) in enumerate(CLASSES)]
87  csv.write("\n".join(rows))
88  csv.close()
```

Each line contains both (1) the human readable class label and (2) a unique integer ID. To build the logos dataset, open up a terminal and execute the following command:

```
$ python build_logos.py
[INFO] starting 'train' set...
[INFO] 748 total images in 'train' set
[INFO] starting 'test' set...
[INFO] 187 total images in 'test' set
[INFO] writing classes...
```

Provided that the command executes successfully, you are ready to move on to training your logo detector using Keras and RetinaNet.

1.4 Training RetinaNet to Detect Logos

To train RetinaNet on the logos dataset you need to use either the `retinanet-train` command or `train.py` script included with the Keras RetinaNet repository. Your `retinanet-train` command should follow the template below:

```
$ retinanet-train --batch-size BATCH_SIZE --steps NUM_STEPS \
     --epochs NUM_EPOCHS --weights COCO_WEIGHTS_PATH \
     --snapshot-path SNAPSHOT_DIRECTORY \
    csv TRAIN.CSV CLASSES.CSV
```

In particular, you'll need to supply the:

- Batch size
- Number of steps per epoch
- Total number of epochs to train for
- Path to the COCO weights file (optional, only supplied when fine-tuning)
- Path to the output snapshot directory to store serialized model weights after each epoch
- And finally, the paths to the training and classes CSV files

Using my Titan X GPU, I was able utilize a batch size of four; however, you may need to tweak this value to either one or two depending on memory limitations of your GPU.

Once you have determined your batch size, you need to compute the number of steps per epoch. This can be done by counting the number of annotations in your `retinanet_train.csv` file:

```
$ wc -l logos/retinanet_train.csv
1393 logos/retinanet_train.csv
```

The `wc` command reveals their are 1,393 lines in the `retinanet_train.csv` file. Since each line represents a single annotation, there are thus 1,393 annotations in the file. With a batch size of four, I can now compute the number of steps per epoch:

$$1393/4 = 348.25 \tag{1.1}$$

Rounding up, there are a total of 349 steps per epoch. Given these values, I was ready to start training with the following command:

```
$ mkdir logos/snapshots
$ retinanet-train --batch-size 4 --steps 349 --epochs 50 \
    --weights logos/resnet50_coco_best_v2.1.0.h5 \
    --snapshot-path logos/snapshots \
    csv logos/retinanet_train.csv logos/retinanet_classes.csv
Epoch 00001: saving model to logos/snapshots/resnet50_csv_01.h5
Epoch 2/50
349/349 [==============================] - 418s 1s/step - loss: 1.7273
     - regression_loss: 0.9701 - classification_loss: 0.7572

Epoch 00002: saving model to logos/snapshots/resnet50_csv_02.h5
Epoch 3/50
349/349 [==============================] - 414s 1s/step - loss: 1.2842
     - regression_loss: 0.7714 - classification_loss: 0.5128

Epoch 00003: saving model to logos/snapshots/resnet50_csv_03.h5
...
Epoch 49/50
349/349 [==============================] - 420s 1s/step - loss: 0.0429
     - regression_loss: 0.0408 - classification_loss: 0.0021

Epoch 00049: saving model to logos/snapshots/resnet50_csv_49.h5

Epoch 00049: ReduceLROnPlateau reducing learning
     rate to 9.999999747378752e-07.
Epoch 50/50
349/349 [==============================] - 416s 1s/step - loss: 0.0319
```

```
      - regression_loss: 0.0306 - classification_loss: 0.0013

Epoch 00050: saving model to logos/snapshots/resnet50_csv_50.h5
```

I let my network train for a total of 50 epochs. On a single Titan X GPU, the training process took ≈ 5.7 hours.

By the end of the 50th epoch I had a low loss value (0.0319) — keep in mind that we normally seek a loss value of < 1.0 when training object detectors as we discussed when using the TFOD API in Chapter 16 of the *ImageNet Bundle*, so our low loss value is quite good.

1.4.1 Exporting Our Model

Before we can evaluate our RetinaNet model or apply it to predict objects in our own images, we first need to *export* the model. Exporting our model can be accomplished by using the `retinanet-convert-model` command:

```
$ retinanet-convert-model logos/snapshots/resnet50_csv_50.h5 output.h5
```

After exporting, we can now use the `output.h5` file for evaluation and prediction (the `resnet50_csv_50.h5` file by itself cannot be used).

1.4.2 Evaluating Our Model

To evaluate our model on a testing set or holdout set, we can use the `retinanet-evaluate` command:

```
$ retinanet-evaluate csv logos/retinanet_test.csv \
     logos/retinanet_classes.csv output.h5
13 instances of class Amazon with average precision: 0.6029
13 instances of class Facebook-Text with average precision: 0.4449
23 instances of class Allianz-Pict with average precision: 0.7158
4 instances of class Atletico_Madrid with average precision: 1.0000
30 instances of class CocaCola with average precision: 0.7501
1 instances of class Facebook-Pict with average precision: 1.0000
21 instances of class Adidas-Text with average precision: 0.3816
25 instances of class Adidas-Pict with average precision: 0.3817
18 instances of class BMW with average precision: 0.8689
16 instances of class Audi-Pict with average precision: 0.5491
12 instances of class Ferrari-Pict with average precision: 0.5417
19 instances of class Aldi with average precision: 0.6374
15 instances of class Ferrari-Text with average precision: 0.2471
23 instances of class FC_Bayern_Munchen with average precision: 0.8403
16 instances of class FC_Barcelona with average precision: 0.5733
5 instances of class Audi-Text with average precision: 0.0000
17 instances of class Burger_king with average precision: 0.9131
13 instances of class Apple with average precision: 0.8863
22 instances of class Allianz-Text with average precision: 0.6646
36 instances of class eBay with average precision: 0.5967
mAP: 0.6298
```

Here you can see that `retinanet-evaluate` command gives us the average precision for *each class* as well as the total *mean average precision* (mean of all the individual average precisions).

Overall, we are obtaining **62.98% mAP** which is quite good given our limited amount of training data. The only logo our model really struggled with was the Audi-Text logo which the RetinaNet model failed to ver detector. However, other class labels such as Atletico Madrid and Facebook were always detected.

1.4.3 Detecting Logos in Test Images

Now that our model is trained, let's apply it to example images not part of the training set. To accomplish this process, open up a new file, name it `predict.py`, and insert the following code:

```
1  # import the necessary packages
2  from keras_retinanet.utils.image import preprocess_image
3  from keras_retinanet.utils.image import read_image_bgr
4  from keras_retinanet.utils.image import resize_image
5  from keras_retinanet import models
6  import numpy as np
7  import argparse
8  import cv2
```

Lines 2-8 import our required Python packages. **Lines 3 and 4** import Keras RetinaNet-specific functions used for loading our example image and pre-processing them.

Next, let's parse our command line arguments:

```
10 # construct the argument parse and parse the arguments
11 ap = argparse.ArgumentParser()
12 ap.add_argument("-m", "--model", required=True,
13      help="path to pre-trained model")
14 ap.add_argument("-l", "--labels", required=True,
15      help="path to class labels")
16 ap.add_argument("-i", "--image", required=True,
17      help="path to input image")
18 ap.add_argument("-c", "--confidence", type=float, default=0.5,
19      help="minimum probability to filter weak detections")
20 args = vars(ap.parse_args())
```

The `--model` switch controls the path to our serialized RetinaNet model. We need to the supply the `--labels` switch, the path to our CSV labels file, in order to obtain the human readable label for a detection. We also need the path to the input `--image` that we are applying object detection to. Finally, we can optionally supply `--confidence`, the minimum probability required for a detection to be marked as such — this value can be used to filter out weak detections.

Now that our command line arguments have been pared, let's load our class label mappings and serialized model form disk:

```
22 # load the class label mappings
23 LABELS = open(args["labels"]).read().strip().split("\n")
24 LABELS = {int(L.split(",")[1]): L.split(",")[0] for L in LABELS}
25
26 # load the model from disk
27 model = load_model(args["model"], custom_objects=custom_objects)
```

Lines 23 and 24 loads the contents of our `--labels` CSV file and then builds a dictionary to map the *unique integer ID* of the prediction (the key) to the *human readable class label* (the value). **Line 27** then loads our serialized RetinaNet from disk.

Next, let's load our image from disk and pre-process it:

```
29 # load the input image (in BGR order), clone it, and preprocess it
30 image = read_image_bgr(args["image"])
31 output = image.copy()
32 image = preprocess_image(image)
33 (image, scale) = resize_image(image)
34 image = np.expand_dims(image, axis=0)
```

In particular, you'll want to examine **Line 34** which resizes our input image. This function will resize the `image` to ensure it has a maximum dimension of 1,333px and a minimum dimension of 800px. You should refer to the RetinaNet GitHub repo [1] for how this scaling takes place. Also note how the `scale` is returned to us so we can use it to rescale our bounding boxes to the original image dimensions in the next code block.

At this point we are now ready to perform object detection using our `model`:

```
36 # detect objects in the input image and correct for the image scale
37 (boxes, scores, labels) = model.predict_on_batch(image)
38 boxes /= scale
```

Line 37 calls the `predict_on_batch` method of our `model` to return:

1. `boxes`: The set of bounding boxes returned by RetinaNet.
2. `scores`: The corresponding probability of the label for each bounding box.
3. `labels`: The predicted labels for each bounding box.

We divide the `boxes` coordinates by the `scale` to rescale them to the dimensions of our original input image (versus the dimensions of the image after applying `preprocess_image`).

The last step is to loop over our bounding boxes + associated class labels and draw them on our image:

```
40 # loop over the detections
41 for (box, score, label) in zip(boxes[0], scores[0], labels[0]):
42     # filter out weak detections
43     if score < args["confidence"]:
44         continue
45
46     # convert the bounding box coordinates from floats to integers
47     box = box.astype("int")
48
49     # build the label and draw the label + bounding box on the output
50     # image
51     label = "{}: {:.2f}".format(LABELS[label], score)
52     cv2.rectangle(output, (box[0], box[1]), (box[2], box[3]),
53         (0, 255, 0), 2)
54     cv2.putText(output, label, (box[0], box[1] - 10),
55         cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0, 255, 0), 2)
56
57 # show the output image
58 cv2.imshow("Output", output)
59 cv2.waitKey(0)
```

On **Line 41** we loop over each of the detections. **Lines 43 and 44** check the probability (i.e., `score`) associated with the detection. If the probability is below our minimum confidence, we discard it — we perform this process to filter out weak predictions.

Line 47 extracts the bounding box coordinates for the current prediction. We then draw the label and bounding box on the `output` image on **Lines 51-55**. Finally, **Lines 58 and 59** display the output image to our screen.

To execute our script, open up a terminal and execute the following command:

```
$ python predict.py --model output.h5 --labels logos/retinanet_classes.csv \
      --image logos/images/786414.jpg --confidence 0.5
```

Figure 1.1: A sample of our logo detection results using Keras RetinaNet. Notice how our network can both correctly *detect* and *label* the logos in images, even when the logo is rotated or skewed.

A sample of the logo detection results can be seen in Figure 1.1. Notice how our network is able to correctly detect and label each of the logos, even when the logo itself is rotated ninety degrees. Unlike traditional Haar cascades or HOG + Linear SVM detectors, our deep learning-based object detectors are not limited by fixed aspect ratios or rotations. Instead, CNNs are naturally able to learn features that activate when the logos appear in a rotated or skewed manner. Provided you have enough labeled data, deep learning-based object detection will likely outperform traditional object detectors.

1.5 Summary

In this unreleased, unpublished bonus chapter to *Deep Learning for Computer Vision with Python* you learned about the RetinaNet object detector. Introduced by Lin et al. in their 2017 paper, *Focal Loss for Dense Object Detection* [3], RetinaNet can be considered a cousin to the R-CNN family (Girshick is listed as an author on the Lin paper) *and* the SSD family of object detectors.

We then discovered how to use the Keras implementation of RetinaNet [1] to train our own custom logo detector. Our logo detector worked extremely well on our sample images, but unfortunately, I could not find a way to ensure the validation and evaluation scripts executed properly.

RetinaNet is more than suitable for use in your own applications; however, be careful when training your network and be sure to perform *a lot* of visual validation to ensure your network is not overfitting.

2. Weapon Detection with RetinaNet

In our last chapter we learned about the RetinaNet [3], a deep learning-based object detection architecture that seeks to combine both the *speed* of one-stage detectors (ex., YOLO and SSD [7, 8]) with the *accuracy* of slower two-stage detectors (ex., Faster R-CNN [9]. After exploring the RetinaNet architecture we then trained a Keras + RetinaNet implementation [1] to detect popular brand logos in images.

In this chapter we're going to explore a case study where we use the RetinaNet chapter to train a network to perform *weapon detection*, and more specifically *handgun detection*, in images and video streams.

Being able to detect weapons and guns in images and video has many applications, including public safety, security systems, and the military sector. As we'll see, we'll be able to reuse much of our code from the previous chapter making it fast and efficient for us to create our weapon detector.

Let's go ahead and get started creating our weapon detector with RetinaNet.

2.1 The Weapons Dataset

The dataset we are going to use in this case study is a collection of handgun images, procured and published by Olmos et al. in their 2018 publication, *Automatic handgun detection alarm in videos using deep learning* [10]. A sample of the handgun images can be seen in Figure 2.1.

The dataset itself consists of 3,000 images containing handguns with a total of 3,463 annotations in PASCAL VOC format, making it easy for us to parse the annotations using a simple XML parser. We will be using this dataset to train our own handgun detector using Keras and RetinaNet.

2.1.1 How Do I Access the Weapons Dataset?

The original weapons dataset can be found on the official project paper from Olmos et al. [10]. However, I found working with their original directory structure a bit tedious and confusing.

To make it easier for you to get started training your own weapon detector, I have kept all images and annotation files as Olmos et al., but made it easier to work it. The following link will take you to the companion website of this book where you can download my modified weapons dataset: http://pyimg.co/gdxyi.

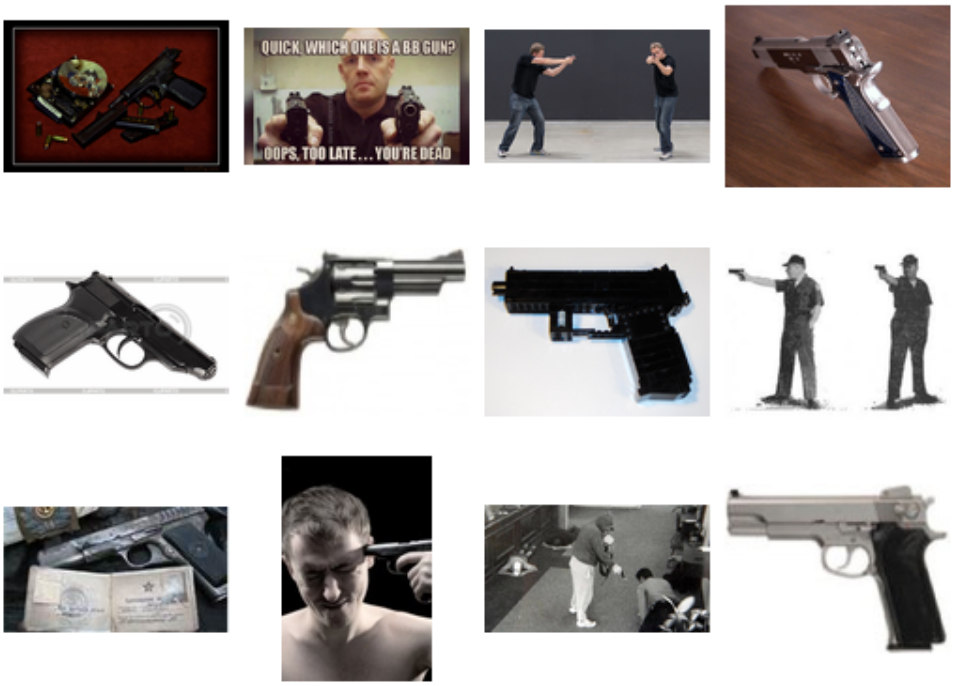


Figure 2.1: A sample of the 3,000 image pistoal dataset [10]. Our goal is to train a RetinaNet model to (1) correctly label the guns and (2) provide bounding box (x, y)-coordinates for each gun in each image.

2.1.2 Project Structure

Before we get too far into this project, let's first review the directory structure:

```
|--- retinanet_weapons
|    |--- config
|    |    |--- __init__.py
|    |    |--- weapons_config.py
|    |--- weapons
|    |    |--- annotations
|    |    |--- images
|    |--- build_weapons.py
|    |--- predict.py
```

The directory structure should look familiar to you as it's near identical to our previous chapter on logo detection.

We'll store all necessary configurations to create our dataset and train our model inside the `weapons_config.py` of the `config` module.

The `weapons` directory contains our actual weapons dataset. Inside the `weapons` directory you'll see our `images` subdirectory, which contains the images themselves, along with the `annotations` directory, which stores an XML file for *each* image. This XML file contains the location of every object (in this case, a handgun) for each respective image.

Take the time now to explore the dataset using your command line. In particular, I recommend listing the contents of both `images` and `annotations`, noting that the filenames in each subdirectory

are near identical except that the files in one subdirectory ends in `.jpg` and in the other subdirectory all files end in `.xml`.

The `build_weapons.py` script will be used to ingest our images and corresponding annotations and then output a CSV compatible with the Keras RetinaNet implementation. We will be implementing this script by hand in Section 2.1.4.

After training our network, we can use the `predict.py` file to find weapons in input images — this file is identical to the `predict.py` file from Chapter 1.

2.1.3 The Configuration File

Our configuration file for the weapons dataset is near identical to the configuration file for the logos dataset in Chapter 1. Open up the `weapons_config.py` file and take a look:

```
1  # import the necessary packages
2  import os
3
4  # initialize the base path for the logos dataset
5  BASE_PATH = "weapons"
6
7  # build the path to the annotations and input images
8  ANNOT_PATH = os.path.sep.join([BASE_PATH, "annotations"])
9  IMAGES_PATH = os.path.sep.join([BASE_PATH, "images"])
10
11 # define the traing/testing split percentage
12 TRAIN_SPLIT = 0.75
13
14 # build the path to the output training and test .csv files
15 TRAIN_CSV = os.path.sep.join([BASE_PATH, "retinanet_train.csv"])
16 TEST_CSV = os.path.sep.join([BASE_PATH, "retinanet_test.csv"])
17
18 # build the path to the output classes CSV file
19 CLASSES_CSV = os.path.sep.join([BASE_PATH, "retinanet_classes.csv"])
```

On **Line 5** we define the base path to the weapons dataset. The paths to the annotations and images themselves are derived from the `BASE_PATH` on **Lines 8 and 9**.

You'll also see that we are using 75% of our data for training and 25% for testing/validation.

Lines 15-19 are identical to our logos configuration from the previous chapter. Here we are defining the paths to our output training and testing CSV files, along with a separate CSV file for the classes themselves.

2.1.4 Building the Dataset

The weapons annotations are stored in PASCAL VOC format, a very popular format used when annotating images for object detection. A sample of one of the weapons XML files can be seen below:

```
1  <annotation>
2    <folder>Definitiva</folder>
3    <filename>armas (3)</filename>
4    <path>C:\Users\Rob\Desktop\Definitiva\armas (3).jpg</path>
5    <source>
6      <database>Unknown</database>
7    </source>
```

```
    <size>
      <width>1300</width>
      <height>866</height>
      <depth>3</depth>
    </size>
    <segmented>0</segmented>
    <object>
      <name>pistol</name>
      <pose>Unspecified</pose>
      <truncated>0</truncated>
      <difficult>0</difficult>
      <bndbox>
        <xmin>471</xmin>
        <ymin>207</ymin>
        <xmax>613</xmax>
        <ymax>359</ymax>
      </bndbox>
    </object>
</annotation>
```

Here you can see the XML structure of the file. Most notably, you'll want to pay attention to both the `size` and `object` elements.

The `size` elements allow us to derive the spatial dimensions of the image itself *without* having to explicitly load the image from disk. Each `object` element defines the `name` (i.e., label) of the object along with its bounding box coordinates (i.e., `bndbox`).

It's important to note that there can be ***multiple object*** elements per XML file as there can naturally be more than one objects in an image.

Now that we've reviewed the basic XML structure of the weapons annotations files, let's start building our dataset.

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

```
# import the necessary packages
from config import weapons_config as config
from bs4 import BeautifulSoup
from imutils import paths
import random
import os

# initialize the set of classes we have encountered so far
CLASSES = set()

# grab all image paths then construct a training and testing split
# from them
imagePaths = list(paths.list_files(config.IMAGES_PATH))
random.shuffle(imagePaths)
i = int(len(imagePaths) * config.TRAIN_SPLIT)
trainImagePaths = imagePaths[:i]
testImagePaths = imagePaths[i:]

# create the list of datasets to build
datasets = [
     ("train", trainImagePaths, config.TRAIN_CSV),
     ("test", testImagePaths, config.TEST_CSV),
]
```

Lines 2-6 import our required Python packages. Notice how we are importing our weapons configuration and aliasing it as `config` to require less keystrokes when coding.

Line 9 defines all `CLASSES` we have encountered while parsing our XML files. **Lines 13-17** grab the paths to all images in the dataset and then compute the training and testing splits.

Finally, **Lines 20-23** define our `dataset` list, mapping the training and testing splits to both their (1) image paths and (2) output CSV file.

Next, let's loop over our datasets:

```
25 # loop over the datasets
26 for (dType, imagePaths, outputCSV) in datasets:
27     # load the contents of the input data split
28     print("[INFO] creating '{}' set...".format(dType))
29     print("[INFO] {} total images in '{}' set".format(
30         len(imagePaths), dType))
31
32     # open the output CSV file
33     csv = open(outputCSV, "w")
34
35     # loop over the image paths
36     for imagePath in imagePaths:
37         # build the corresponding annotation path
38         fname = imagePath.split(os.path.sep)[-1]
39         fname = "{}.xml".format(fname[:fname.rfind(".")])
40         annotPath = os.path.sep.join([config.ANNOT_PATH, fname])
41
42         # load the contents of the annotations file and build the soup
43         contents = open(annotPath).read()
44         soup = BeautifulSoup(contents, "html.parser")
```

For each of our dataset splits we'll need to create an output CSV file in Keras + RetinaNet format, just as we did in Chapter 1 (**Line 33**).

Each of our dataset splits consists of multiple images, therefore we'll start to loop over each of the image paths on **Line 36**. We'll then derive the filename of the current image, strip off the `.jpg` extension, and then construct the path to the corresponding XML annotation file — the XML file is then loaded from disk on **Line 43**.

Line 44 utilizes the `beautifulsoup` package [11], a highly popular XML parser, to parse the XML file `contents`.

If you have never parsed an XML file before be sure to refer to Chapter 17 of the ImageNet Bundle where I provide resources on where you can learn to parse XML files. The concept itself is simple and won't take you longer than a few minutes to master.

Now that we have our XML file loaded and parsed, we start extracting the bounding box coordinates of each of the objects in the image:

```
46         # extract the image dimensions
47         w = int(soup.find("width").string)
48         h = int(soup.find("height").string)
49
50         # loop over all object elements
51         for o in soup.find_all("object"):
52             # extract the label and bounding box coordinates
```

```
53                  label = o.find("name").string
54                  xMin = int(o.find("xmin").string)
55                  yMin = int(o.find("ymin").string)
56                  xMax = int(o.find("xmax").string)
57                  yMax = int(o.find("ymax").string)
```

Lines 47 and 48 extract the width and height of the current image. We then loop over all `object` elements in the XML file (**Line 51**), extracting both the `label` (**Line 53**) and bounding box coordinates as we go (**Lines 57**).

Given the bounding box coordinates we need to perform a few sanity checks:

```
59                  # truncate any bounding box coordinates that may fall outside
60                  # the boundaries of the image
61                  xMin = max(0, xMin)
62                  yMin = max(0, yMin)
63                  xMax = min(w, xMax)
64                  yMax = min(h, yMax)
65
66                  # due to errors in annotation, it may be possible that
67                  # the minimum values are larger than the maximum values;
68                  # in this case, treat it as an error during annotation
69                  # and ignore the bounding box
70                  if xMin >= xMax or yMin >= yMax:
71                       continue
72
73                  # similarly, we could run into the opposite case where
74                  # the max values are smaller than the minimum values
75                  elif xMax <= xMin or yMax <= yMin:
76                       continue
```

In the case that any bounding box coordinates may fall outside the boundaries of the image, we truncate them (**Lines 61-64**).

Furthermore, we'll also ignore any annotations that cannot make logical sense, such as the minimum x or y coordinates being larger than than their maximum and vice versa (**Lines 70-76**).

Provided that our bounding box coordinates are valid, we cannot write them to disk in CSV format:

```
78                  # write the image path, bounding box coordinates, and label
79                  # to the output CSV file
80                  row = [os.path.abspath(imagePath), str(xMin), str(yMin),
81                       str(xMax), str(yMax), label]
82                  csv.write("{}\n".format(",".join(row)))
83
84                  # update the set of unqiue class labels
85                  CLASSES.add(label)
86
87       # close the CSV file
88       csv.close()
89
90  # write the classes to file
91  print("[INFO] writing classes...")
92  csv = open(config.CLASSES_CSV, "w")
```

```
93 rows = [",".join([c, str(i)]) for (i, c) in enumerate(CLASSES)]
94 csv.write("\n".join(rows))
95 csv.close()
```

For the exact format of the CSV row, please refer to Chapter 1 where I have discussed the format in detail.

Line 85 adds our current `label` to the set of all `CLASSES` in our dataset while **Line 88** closes the CSV file for the current data split.

Finally, **Lines 92-95** dumps the `CLASSES` to disk.

2.2 Training RetinaNet to Detect Weapons

To train RetinaNet on our weapons dataset you can use either the `retinanet-train` command or the `train.py` script included with the Keras RetinaNet repository. You'll want to refer to Chapter 1, Section 1.4 where I discuss the format of the command in detail.

However, before we can train our network we first need to determine the *batch size* along with the *number of batches per epoch.* With my Titan X GPU, I have elected to use a batch size of 2.

Next, I need to determine the number of batches per epoch. This task can be accomplished by counting the number of annotations in your `retinanet_train.csv` file:

```
$ wc -l weapons/retinanet_train.csv
2618 weapons/retinanet_train.csv
```

Here you can see the `wc` command indicates there are 2,618 images in the dataset. Using this information I can now derive the steps per epoch:

$$2618/2 = 1309 \tag{2.1}$$

Given these values I can now start training:

```
$ retinanet-train --batch-size 2 --steps 1309 --epochs 50 \
    --weights weapons/resnet50_coco_best_v2.1.0.h5 \
    --snapshot-path weapons/snapshots \
    csv weapons/retinanet_train.csv weapons/retinanet_classes.csv
Epoch 1/50
1309/1309 - 926s 707ms/step - loss: 1.1528 - regression_loss: 0.8215 \
     - classification_loss: 0.3313

Epoch 00001: saving model to weapons/snapshots/resnet50_csv_01.h5
Epoch 2/50
1309/1309 - 796s 608ms/step - loss: 0.7344 - regression_loss: 0.5992 - \
     classification_loss: 0.1352

Epoch 00002: saving model to weapons/snapshots/resnet50_csv_02.h5
...
Epoch 49/50
1309/1309 - 607ms/step - loss: 0.0408 - regression_loss: 0.0401 - \
     classification_loss: 7.0723e-04

Epoch 00049: saving model to weapons/snapshots/resnet50_csv_49.h5
Epoch 50/50
```

```
1309/1309 - 796s 608ms/step - loss: 0.0411 - regression_loss: 0.0405 \
    - classification_loss: 6.1970e-04

Epoch 00050: saving model to weapons/snapshots/resnet50_csv_50.h5
```

I let my network train for a total of 50 epochs. But before I can evaluate my network or apply it to my own images, I first need to convert the model using the following command:

```
$ retinanet-convert-model weapons/snapshots/resnet50_csv_50.h5 output.h5
```

Here you can see that I am converting the snapshot of my RetinaNet object detector with ResNet backbone after the 50th epoch to an output file named `output.h5`.

Finally, I can use the `retinanet-evaluate` command to compute the mAP on the validation dataset:

```
$ retinanet-evaluate csv weapons/retinanet_test.csv \
    weapons/retinanet_classes.csv output.h5
Loading model, this may take a second...
845 instances of class pistol with average precision: 0.8776
mAP: 0.8776
```

After evaluation you can see that our model is obtaining **87.76% mAP** (mean Average Precision) which is *extremely* good.

2.3 Detecting Weapons in Test Images

Let's go ahead and apply your weapon detector to some example images!

To accomplish this task, we'll be using the `predict.py` file which is identical to the prediction script from Chapter 1 — if you need a refresher on how this script works, please refer to Section 1.4.3 where I review the script in its entirety.

To execute the script, open up a terminal and execute the following command:

```
$ python predict.py --model output.h5 --labels weapons/retinanet_classes.csv \
    --image weapons/images/armas_1.jpg --confidence 0.5
```

A sample of the weapon detections can be seen in Figure 2.2. Notice how our RetinaNet model is capable of correctly localizing pistols in images — such a system can be used in home security systems, schools, office places, and other publicly monitored areas where safety is paramount.

2.4 Summary

In this chapter we learned how to use the RetinaNet architecture [3] implemented in Keras [1] to quickly train a weapon object detection system, obtaining over **87.76% mAP.** Such a system can be used for in schools, offices, places of worship, and other publicly monitored areas where safety is of upmost concern.

Furthermore, we were able to reuse much of our code from the previous chapter, requiring only minor updates.

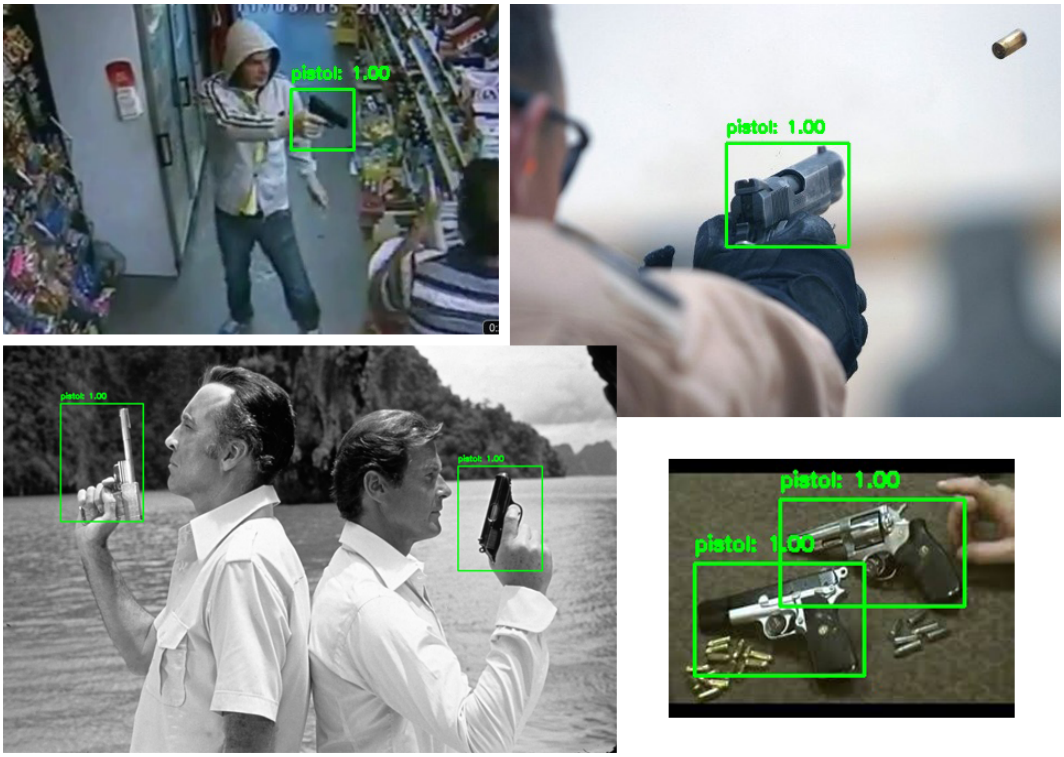


Figure 2.2: Examples of using our RetinaNet object detection model to detect and localize pistols in security footage *(top-left)*, professional photography *(top-right)*, cinema *(bottom-left)*, and crime scene photos *(bottom-right)*.

Whenever you use the Keras implementation of RetinaNet you'll initially spend most of your time massaging your data into the CSV file format that the framework requires. From there, you can move on to training and tuning your hyperparameters.

During training you'll want to intermittently evaluate your network on the validation set by computing the mAP — the Keras + RetinaNet implementation does not automatically perform this computation for you so you'll either want to (1) explicitly run the command yourself or (2) schedule a cronjob that finds the latest output serialized model and then computes the mAP for that model.

3. Mask R-CNN and Cancer Detection

So far in this book we've explored two primary forms of image understanding: *classification* and *detection*. When performing image classification our goal is to predict a single label that characterizes the contents of the entire image.

Object detection on the other hand seeks to (1) find *all* objects in an image, (2) label them, and (3), compute their bounding box (x, y)-coordinates, enabling us to not only determine *what* is in an image but also *where* in the image an object resides.

Object detection is clearly a harder task than image classification. But a task even *harder* than object detection is *segmentation* and more specifically, *instance segmentation.*

Instance segmentation takes object detection a step further — instead of predicting a bounding box for each object in an image, we now see to *predict a mask* for each object, giving us a pixel-wise segmentation of the object rather than a coarse, perhaps even unreliable bounding box. Instance segmentation algorithms power much of the "magic" we see in computer vision, including self-driving cars, robotics, and more.

In this chapter we will start with a brief discussion of object detection, semantic segmentation, and instance segmentation, including the differences between them.

From there we'll review the Mask R-CNN architecture [12] which can be used for instance segmentation. We'll also explore Keras Mask R-CNN [13], an implementation of the Mask R-CNN framework that we'll be using in this chapter along with the following one.

Once we understand how the Mask R-CNN framework is used for instance segmentation I'll be showing you how to train your own instance segmentation networks for cancer detection, specifically skin lesion analysis for melanoma detection.

3.1 Object Detection vs. Instance Segmentation

Deep learning has helped facilitate unprecedented accuracy in computer vision, including image classification, object detection, and now even *segmentation.* So, what is the difference between these types of algorithms?

As we learned from earlier in the *Practitioner Bundle* and *ImageNet Bundle* of this book we know that object detection is the process of (1) detecting objects in an image followed by

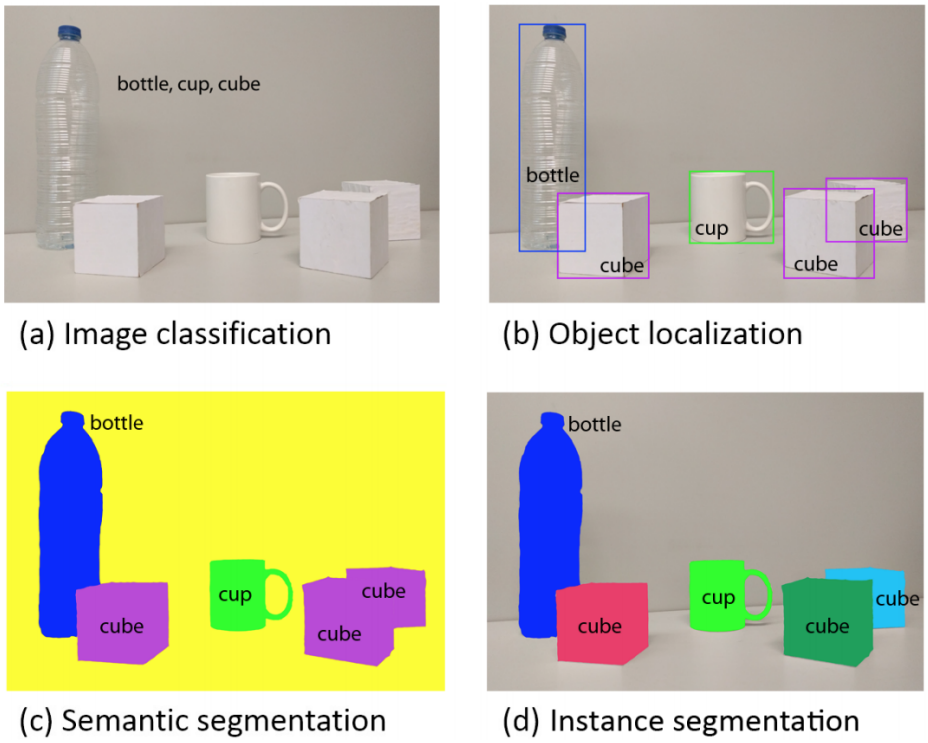


Figure 3.1: **Top-left:** When performing *image classification* we predict a set of labels for an input image. **Top-right:** *Object detection* allows us to predict the bounding box (x, y)-coordinates for *each* object in an image along with the class label. *Bottom-left: Semantic segmentation* associates *every* pixel in an image with a label. **Bottom-right:** *Instance segmentation* computes a pixel-wise mask for *every* object in the image, even if objects are of the same class label. Figure from Garcia-Garcia et al. [14]

(2) predicting their corresponding bounding box coordinates and class labels. An example of performing object detection can be seen in Figure 3.1 *(top-right)*.

Traditional segmentation on the other hand involves partitioning an image into parts. Popular traditional segmentation algorithms include Normalized Cuts [15], Graph Cuts [16], Grab Cuts [17], and superpixels [18, 19, 20, 21]; *however*, none of these algorithms have any actual *understanding* of what each of the resulting segmented image parts represent.

Semantic segmentation (Figure 3.1, *bottom-left*) and instance segmentation (Figure 3.1, *bottom-right*) algorithms seek to introduce *understanding* into their output parts.

Specifically, semantic segmentation algorithms attempt to:

- *Partition* the image into meaningful parts
- While at the same time, *associating every pixel* in an input image with a class label (i.e., person, road, car, bus, etc.).

Semantic segmentation algorithms have many use cases but are most popularly used in self-driving cars where we seek to understand the *entire scene* that a car camera is looking at. Semantic segmentation algorithms may also require that each of the objects, regardless of class, be *uniquely labeled* as well. This type of segmentation is called a *panoptic segmentation*.

It's important to note that panoptic segmentation is an *extremely* challenging problem since we need to ensure that not only *every* pixel in an input image is assigned to a label, but also each object is *uniquely labeled* and segmented as well.

Instance segmentation on the other hand is a some-what easier problem (compared to panoptic segmentation). Instance segmentation can be thought of a hybrid between both object detection and semantic segmentation.

While object detection produces a *bounding box*, instance segmentation produces a pixel-wise *mask* for each individual object object. Instance segmentation does not require every pixel in an image to be associated with a label, however.

Figure 3.1 provides visualizations for image classification *(top-left)*, object detection *(top-right)*, semantic segmentation *(bottom-left)*, and instance segmentation *(bottom-right)*. Since instance segmentation tends to be a more tractable, realistic use case for most problems, and not to mention, there is more labeled data for such a task, we'll be focusing on performing instance segmentation with the Mask R-CNN framework in this chapter along with the following chapter.

3.1.1 What is Mask R-CNN?

The Mask R-CNN algorithm was introduced by He et al. in their 2017 paper, *Mask R-CNN*. Mask R-CNN builds on the previous object detection work of R-CNN, Fast R-CNN, and Faster R-CNN by Girshick et al. [9, 22, 23]

The Faster R-CNN family of object detectors is covered in detail inside Section 14.2 of Chapter 14 of the *ImageNet Bundle* — please refer to this chapter if you need a refresher on how the Faster R-CNN algorithm works as I'll be assuming you have read the chapter before continuing your study of Mask R-CNN.

As we know from our previous study of Faster R-CNN in the *ImageNet Bundle*, the Faster R-CNN architecture takes the output of the ROI Pooling module and passes it through two FC layers (each 4096-d), similar to the final layers of classification networks of ImageNet.

The output of these `FC` layers feed into the final two `FC` layers of the network. One `FC` layer is $N+1$-d, a node for each of the class labels plus an additional label for the background. The second `FC` layer is $4 \times N$ which represents the deltas for final predicted bounding box.

Thus, we can think of a Faster R-CNN as producing two sets of values:

- The class label probabilities
- The corresponding bounding box predictions

A visualization of the final Faster R-CNN component of the architecture can be seen in Figure 3.2 *(top)*.

To turn our Faster R-CNN into a Mask R-CNN, He et al. proposed adding an *additional branch* to the network. This additional branch is responsible for predicting the actual *mask* of an object/class. A visualization of the updated architecture can be see in Figure 3.2 *(bottom)*.

Notice how a new parallel branch has been introduced in the network architecture. The masking branch splits off from the ROI Align module (an improvement to the Girshick et al. ROI Pool module) *prior* to our `FC` layers and then consists of two `CONV` layers responsible for creating the mask predictions themselves.

In practice, Mask R-CNN introduces only a small overhead to the Faster R-CNN framework and can still run at approximately 5 FPS with a. ResNet-101 backend on a GPU [12]. Furthermore, just like Faster R-CNN and SSD can leverage different architectures such as ResNet, VGG, SqueezeNet, MobileNet, etc. as their backend/backbone, making it possible to decrease mobile size and potentially increase FPS throughput as well.

Understanding Mask R-CNN Output Volume Size

Before we get too far into this chapter I believe it's worth having a discussion on the output volume dimensions of a Mask R-CNN.

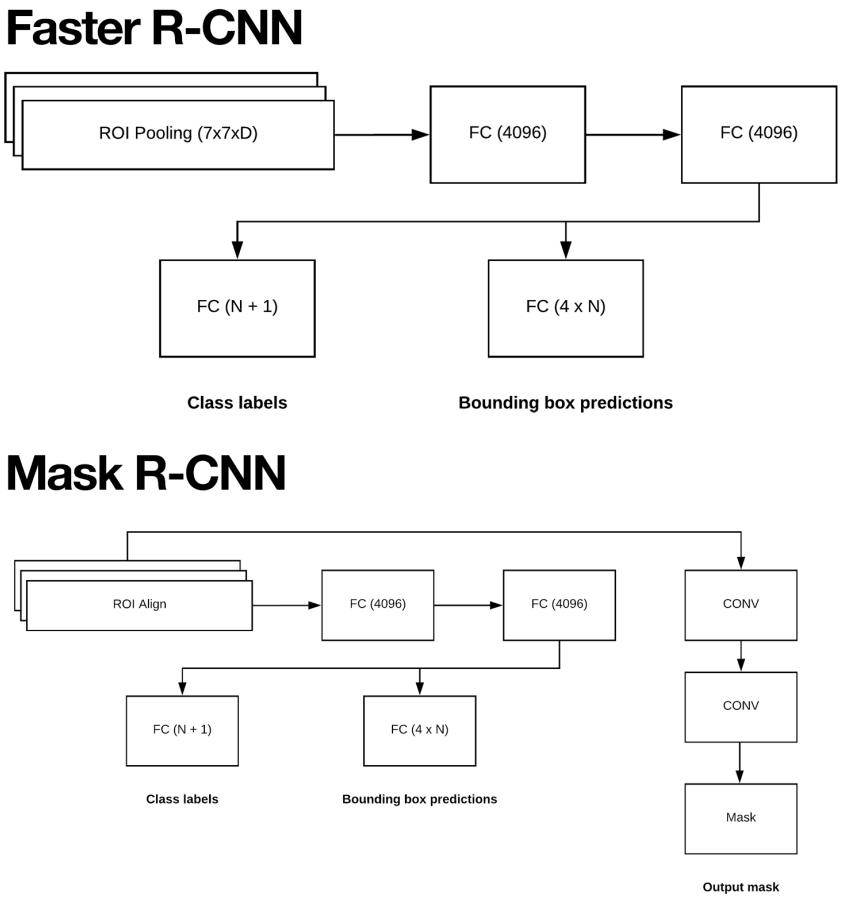


Figure 3.2: **Top:** The original output layers of the Faster R-CNN framework [9]. One `FC` layer contains our bounding box offset predictions while the second `FC` layer contains the corresponding class labels for each bounding box. **Bottom:** The Mask R-CNN work by [12] replaces the ROI Pooling module with a more accurate ROI Align module. The output of the ROI module is then fed into two `CONV` layers. The output of these `CONV` layers is the mask itself.

As we know from Chapter 14 of the *ImageNet Bundle*, the Faster R-CNN/Mask R-CNN leverages a Region Proposal Network (RPN) to generate regions of an image that *potentially* contain an object.

Each of these regions is ranked based on their "objectness score" (i.e., how likely it is a given region could potentially contain an object) and then the top N most confident objectness regions are kept.

In the original Faster R-CNN publication, Girshick et al. set $N = 2,000$, but in practice we can get away with a much smaller N, such as $N = 10, 100, 200, 300$ and still obtain good results. He et al. set $N = 300$ in their publication which is the value we'll use here as well.

Each of the 300 most ROIs then go through the three parallel branches of the network:

- Label prediction
- Bounding box prediction
- Mask prediction

Figure 3.2 from earlier in this chapter provides a visualization of each of these branches.

During prediction, each of the 300 ROIs go through non-maxima suppression [24] and the top 100 detection boxes are kept, resulting in a 4D tensor of $100 \times L \times 15 \times 15$ where L is the number

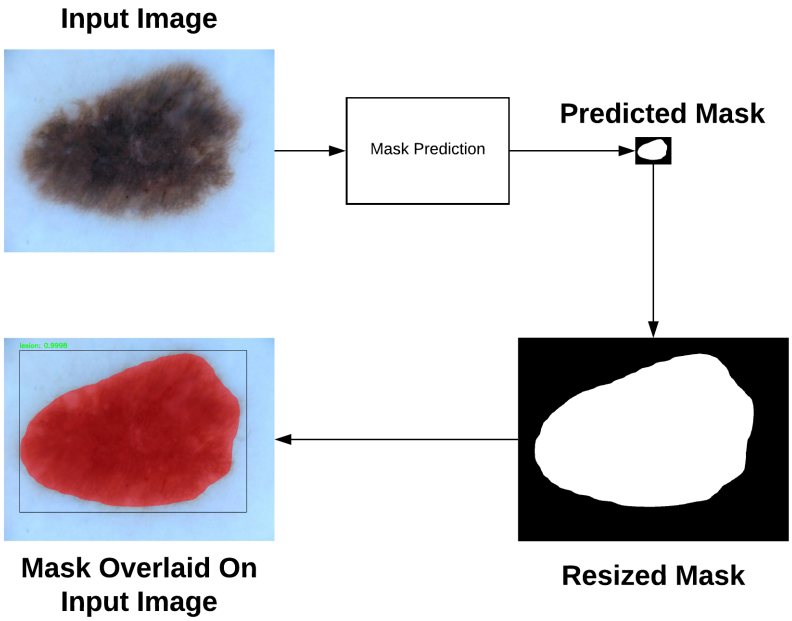


Figure 3.3: We start with our input image and then feed it through our Mask R-CNN network to obtain our mask prediction. The predicted mask is only 15×15 pixels so we apply nearest neighbor interpolation to resize it back to the original image dimensions. The resized mask can then be overlaid on the original input image.

of class labels in the dataset and 15×15 is the size of each of the L masks.

For example, if we're using the COCO dataset, which has $L = 90$ classes, then the resulting volume size from the mask module of the Mask R-CNN will be $100 \times 90 \times 15 \times 15$.

After performing a forward pass we can loop over each of the detected bounding boxes, find the class label index with the largest corresponding probability, and then use the index lookup the 15×15 mask in the $100 \times 90 \times 15 \times 15$ volume.

Written in pseudocode our lookup may look like: `instanceMask = masks[i, classID]`, where i is the index of the current detection (i.e., the first dimension of the output volume) and `classID` is the index of the class label with the largest corresponding probability.

After performing the lookup we now have `instanceMask`, a 15×15 mask that represents the pixel-wise segmentation of the i-th detected object.

We can then scale the mask back to the original dimensions of the image *making sure to use nearest neighbor interpolation*, allowing us to obtain the pixel-wise mask for the original image.

An example of extracting a 15×15 mask, resizing to the original image dimensions, and then overlaying the mask on the detected object can be seen in Figure 3.3.

So, why nearest neighbor interpolation? A discussion of interpolation algorithms is well outside the scope of this book and, is not to mention, a field of study itself; however, the gist is that nearest neighbor interpolation will preserve all original values of the image during resizing without introducing new values.

While nearest neighbor interpolation is less appeasing to the human eye, such as bilinear or bicubic interpolation, for example, the benefit here is that we do not introduce "new" object/mask assignments when resizing the image. Make sure when resizing your own masks that you are performing nearest neighbor interpolation as well.

For a more detailed review of interpolation algorithms, including visual examples, please refer to the following excellent article on MathWorks: http://pyimg.co/k8r5r.

Additionally, if you would like more information on Mask R-CNN, including more visual examples on volume size and how we use to derive the resulting mask for an object, please refer to the *Mask R-CNN with OpenCV* blog post on the PyImageSearch site: http://pyimg.co/4lwab.

I would *highly encourage* you to read the blog post as it will give you more exposure to the Mask R-CNN architecture (namely how to process output predictions), making it easier for you to grasp, comprehend, and retain the information in the rest of this chapter.

3.2 Installing Keras Mask R-CNN

To actually train our own custom Mask R-CNN models, we'll be using the Matterport Keras + Mask R-CNN implementation [13]. The Matterport implementation is arguably one of the most used and stable Keras implementations of Mask R-CNN.

Furthermore, the framework is fairly intuitive, but there are a few aspects that can and will trip you up as you get started with the library. I learned first-hand with this library and will be sharing my experience along the way to ensure you do not run into the same problems I did.

Just like RetinaNet, the Keras Mask R-CNN implementation is still considered "under development" with no official releases. Since libraries and packages under development can and will change frequently, I've decided to move the installing and configuration section to the companion website where I can more easily update the install instructions as they change.

To install the Keras Mask R-CNN library on your machine, please refer to this link: http://pyimg.co/uib16.

And if you do not already have an account on the companion website, please refer to the first few pages of this book where we'll find a link to create your companion website account.

3.3 The ISIC Skin Lesion Dataset

The dataset we'll be using in this case study is the International Skin Imaging Collaboration (ISIC) 2018 Skin Lesions dataset [25, 26]. This particular dataset enables computer vision researchers, developers, and engineers to contribute to the study and early detection of cancer, in particular melanoma.

Melanoma is a major public health concern with over 5,000,000 newly diagnosed cases per year in the United States alone. Melanoma is also the deadliest form of skin cancer with 350,000 reported cases and 60,000 deaths in 2015.

The good news is that when caught early, melanoma survival rates are in excess of 95%. Therefore, early detection of the cancer is key.

In mid-2018, ISIC released a three part imaging challenge, including:

- Lesion segmentation
- Lesion attribute detection
- Disease classification

The goal of lesion segmentation is to automatically segment the boundaries of potential cancerous regions from dermoscopic images — this challenge lends itself *perfectly* to instance segmentation and is the one we will be examining in this case study.

You can find an example of skin lesion boundary detection in Figure 3.4. The *top* row contains examples of skin lesion images while the *bottom* row provides their corresponding masks. Our goal will be to correctly predict those pixel-wise masks from the input image.

The second task is more challenging, involving automatically predicting and localizing various attributes of the lesion, such as pigment networks, streaks, milia-like cysts, and more.

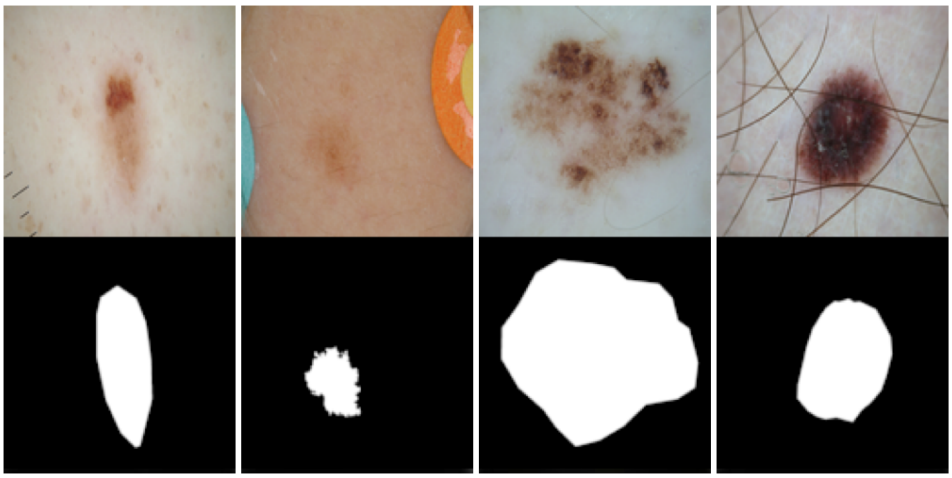


Figure 3.4: The dataset we'll be using for this chapter is from the International Skin Imaging Collaboration (ISIC) 2018 Skin Lesions challenge [25, 26]. We are provided with the skin lesion images *(top)* along with their corresponding masks (*bottom*). Our goal is to train a Mask R-CNN to accurately predict such masks for new skin lesion inputs. These masks can aid a computer vision-based system in predicting and classifying melanoma, a type of skin cancer.

The final task is simple image classification — given an input image the goal is to classify the lesion into seven distinct categories such as Melanoma, Basal cell carcinoma, Melanocytic nevus, and others.

You can learn more about the ISIC 2018 dataset, including more information regarding each of the three challenges, from their official challenge page (http://pyimg.co/u3nuv). In the context of the rest of this chapter, we'll be focusing primarily on the first task — skin lesion boundary detection.

3.3.1 How Do I Download the ISIC 2018 Dataset?

The ISIC 2018 dataset is freely available for you to use and study from for non-commercial purposes. To access the dataset you first need to create an account on the ISIC 2018 website: http://pyimg.co/h7uxv.

After registering you'll be able to click the *"Download training dataset"* and *"Download ground truth data"* buttons to download .zip archives of the training data and corresponding ground-truth masks (Figure 3.5).

3.3.2 Skin Lesion Boundary Detection Challenge

The skin lesion boundary segmentation dataset consists of 2,594 images in JPEG format (11GB) and 2,594 corresponding ground-truth masks in PNG format (27MB).

Why PNG? PNG is a *lossless* image file format, implying that the data compression algorithm used for PNG images allows the original data to be 100% *perfectly* reconstructed from compressed data.

Lossy file formats on the other hand, such as JPEG, can reduce the actual size of our images, and most of the time without our eyes being able to tell the difference.

The reason we use lossless image file formats for the mask is to (1) ensure the pixel-wise segmentation of the mask is perfectly retained and (2) that any pixel values themselves are perfectly retained. If we used a lossless file format those pixel values may change slightly, perhaps rendering

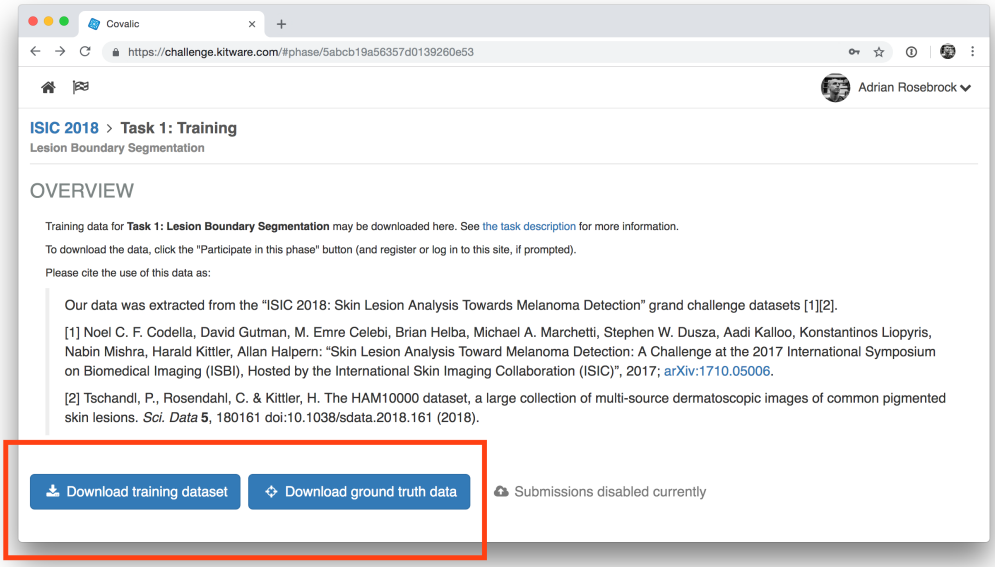


Figure 3.5: After registering and logging in, you will be able to download bout the training images and corresponding ground-truth masks.

our masks unusable.

All masks are encoded as single-channel (grayscale) images. Each pixel in the mask has only one of two values:

1. `0`: Areas *outside* the lesion (i.e., background)
2. `255`: Areas *inside* the lesion (i.e., foreground)

After unzipping your two training data you should have the following directory structure:

```
$ ls -l
ISIC2018_Task1-2_Training_Input
ISIC2018_Task1-2_Training_Input.zip
ISIC2018_Task1_Training_GroundTruth
ISIC2018_Task1_Training_GroundTruth.zip
```

Notice how the training images and ground-truth directories are clearly named as such.

Let's take a look at the filenames of our training images:

```
$ ls -l ISIC2018_Task1-2_Training_Input/*.jpg | head -n 5
ISIC2018_Task1-2_Training_Input/ISIC_0000000.jpg
ISIC2018_Task1-2_Training_Input/ISIC_0000001.jpg
ISIC2018_Task1-2_Training_Input/ISIC_0000003.jpg
ISIC2018_Task1-2_Training_Input/ISIC_0000004.jpg
ISIC2018_Task1-2_Training_Input/ISIC_0000006.jpg
```

Note how all input images have the format `ISIC_<image_id>.jpg` where `image_id` is a 7-digit unique identifier. We can use this 7-digit ID to associate our input images to their corresponding masks.

Our mask images have a similar filename structure:

```
$ ls -l ISIC2018_Task1_Training_GroundTruth/*.png | head -n 5
ISIC2018_Task1_Training_GroundTruth/ISIC_0000000_segmentation.png
ISIC2018_Task1_Training_GroundTruth/ISIC_0000001_segmentation.png
ISIC2018_Task1_Training_GroundTruth/ISIC_0000003_segmentation.png
ISIC2018_Task1_Training_GroundTruth/ISIC_0000004_segmentation.png
ISIC2018_Task1_Training_GroundTruth/ISIC_0000006_segmentation.png
```

Notice the only difference between the input images and their corresponding masks is that the mask images (1) include `_segmentation` and the end of their filename and (2) are a PNG filetype.

Given the similarities between the filenames it will be uncomplicated and straightforward for us to derive the mask filename from an input image filename and vice versa.

3.4 Training Your Mask R-CNN

Now that we've explored our skin lesion dataset, let's start exploring how we can train a Mask R-CNN to predict the boundaries of the lesion areas.

3.4.1 Project Structure

Before we can start training our Mask R-CNN, we first need to examine our project structure:

```
|--- mask_rcnn
|    |--- isic2018/
|    |    |--- ISIC2018_Task1-2_Training_Input
|    |    |--- ISIC2018_Task1_Training_GroundTruth
|    |--- lesions.py
|    |--- mask_rcnn_coco.h5
|    |--- mrcnn/
```

The `isic2018` directory contains the ISIC 2018 dataset itself. You can either store the unzipped directories from Section 3.3.2 in this directory or you can do what I did and create a sym-link to the dataset via the `ln` command — either will achieve the desired result.

The `lesions.py` file contains all code that we will need to train our Mask R-CNN. We will be implementing the code in this file in the next section.

The `mask_rcnn_coco.h5` file is a Mask R-CNN with ResNet backbone pre-trained on the COCO dataset [5]. We will be fine-tuning this model for our own segmentation tasks.

In nearly all situations it is wise to start with a *pre-trained* model and the *fine-tune* it — training from scratch not only takes longer, but is more tedious, typically requiring additional training data as well. I have included the `mask_rcnn_coco.h5` file in the downloads associated with this book but you can also download the file here: http://pyimg.co/r5992.

Finally, the `mrcnn` directory contains the Matterport Keras + Mask R-CNN implementation.

If you followed by Keras + Mask R-CNN install sections from Section 3.2 then you should have cloned the Matterport Mask R-CNN GitHub repository into your home directory. From there, you can either (1) copy the `Mask-RCNN/mrcnn` directory into this project or (2) do what I do and create a sym-link:

```
$ ln -s ~/Mask_RCNN/mrcnn mrcnn
```

Now that we've reviewed our project structure, let's go ahead and implementing the training script!

3.4.2 Implementing the Mask R-CNN

To get started, open up the `lesions.py` file and insert the following code:

```
1  # import the necessary packages
2  from imgaug import augmenters as iaa
3  from mrcnn.config import Config
4  from mrcnn import model as modellib
5  from mrcnn import visualize
6  from mrcnn import utils
7  from imutils import paths
8  import numpy as np
9  import argparse
10 import imutils
11 import random
12 import cv2
13 import os
```

Lines 2-13 handle importing our required Python packages and libraries. Notice how **Line 2** imports from the `imgaug` library [27].

As we know from chapters in both the *Practitioner Bundle* and *ImageNet Bundle*, image augmentation allows us to create additional training data on the fly by applying random transformations to our input images, including rotation, translation, scaling, etc. Image augmentation is often *critical* in reducing overfitting and increasing the ability of your model to generalize.

That said, applying image augmentation to instance segmentation isn't as easy or straightforward as classification — any transformation that is performed to the input *image* also needs to be performed on the *mask* as well.

The `imgaug` library enables us to perform more powerful augmentation than what the standard Keras `ImageDataGenerator` does. For more information on `imgaug`, please refer to their official GitHub repository [27].

Lines 3-6 import our Mask R-CNN classes and functions. We'll be subclassing the `Config` class in order to derive our own configuration for training on the ISIC 2018 dataset.

The `modellib` contains the Mask R-CNN model itself. The `visualize` submodule will help us visualize the output predictions of the Mask R-CNN while the `utils` submodule contains various utilities we'll be leveraging in this script.

Next, let's derive the path to both our images and masks:

```
15 # initialize the dataset path, images path, and annotations file path
16 DATASET_PATH = os.path.abspath("isic2018")
17 IMAGES_PATH = os.path.sep.join([DATASET_PATH,
18      "ISIC2018_Task1-2_Training_Input"])
19 MASKS_PATH = os.path.sep.join([DATASET_PATH,
20      "ISIC2018_Task1_Training_GroundTruth"])
```

The `DATASET_PATH` is the root directory of the ISIC 2018 dataset. We then use the `DATASET_PATH` to derive the path to where all input skin lesion images are stored (`IMAGES_PATH`) as well as their corresponding masks (`MASKS_PATH`).

Given these paths we can then construct our training and validation split:

```
22 # initialize the amount of data to use for training
23 TRAINING_SPLIT = 0.8
```

```
24
25  # grab all image paths, then randomly select indexes for both training
26  # and validation
27  IMAGE_PATHS = sorted(list(paths.list_images(IMAGES_PATH)))
28  idxs = list(range(0, len(IMAGE_PATHS)))
29  random.seed(42)
30  random.shuffle(idxs)
31  i = int(len(idxs) * TRAINING_SPLIT)
32  trainIdxs = idxs[:i]
33  valIdxs = idxs[i:]
```

Line 23 indicates that we are using 80% of our data for training and 20% for validation.

From there, **Line 27** grabs all `IMAGE_PATHS` in the input `IMAGES_PATH` directory.

Line 28 generates an indexes (`idxs`) array with the same length as `IMAGE_PATHS`. We randomly shuffle the indexes (**Line 30**) and apply array slicing (**Lines 31-33**) to determine our `trainIdxs` and `valIdxs`. We use *indexes* rather than the raw file paths for reasons to become apparent later in this core review.

We also need to define a `CLASS_NAMES` dictionary:

```
35  # initialize the class names dictionary
36  CLASS_NAMES = {1: "lesion"}
37
38  # initialize the path to the Mask R-CNN pre-trained on COCO
39  COCO_PATH = "mask_rcnn_coco.h5"
40
41  # initialize the name of the directory where logs and output model
42  # snapshots will be stored
43  LOGS_AND_MODEL_DIR = "lesions_logs"
```

The `CLASS_NAMES` dictionary stores the unique integer ID of the class as the key and the human-readable class label as the value. For the ISIC 2018 skin lesion boundary detection challenge we have only *one* class — the lesion itself.

We also define the path to the pre-trained COCO model (`COCO_PATH`) as well as a directory where all output model snapshots and logs will be stored (`LOGS_AND_MODEL_DIR`).

The Matterport Mask R-CNN implementation attempts to make it as easy as possible for us to train our own Mask R-CNNs. To accomplish this task, they require us to subclass the `Config` class and override and/or define any configurations we may need.

Let's get started by examining our *training* configuration:

```
45  class LesionBoundaryConfig(Config):
46      # give the configuration a recognizable name
47      NAME = "lesion"
48
49      # set the number of GPUs to use training along with the number of
50      # images per GPU (which may have to be tuned depending on how
51      # much memory your GPU has)
52      GPU_COUNT = 1
53      IMAGES_PER_GPU = 1
54
55      # set the number of steps per training epoch and validation cycle
56      STEPS_PER_EPOCH = len(trainIdxs) // (IMAGES_PER_GPU * GPU_COUNT)
57      VALIDATION_STEPS = len(valIdxs) // (IMAGES_PER_GPU * GPU_COUNT)
```

```
58
59      # number of classes (+1 for the background)
60      NUM_CLASSES = len(CLASS_NAMES) + 1
```

The `LesionBoundaryConfig` class stores all relevant configurations when training our Mask R-CNN on the skin lesion dataset. To start, we need to give this dataset and experiment a `NAME` — we aptly name the dataset `lesion`.

Line 52 and 53 define the total number of GPUs we'll be using for training along with the number of images per GPU (i.e., batch size). I performed all experiments for this case study on a machine with a single Titan X GPU so I set my `GPU_COUNT` to one.

While my 12GB GPU could technically handle more than one image at a time, I decided to set `IMAGES_PER_GPU` to one as most readers will not have a GPU with as much memory. Feel free to increase this value if your GPU can handle it.

Line 56 defines our `STEPS_PER_EPOCH` which is simply the number of training images divided by the number of images per GPU and the GPU count. Similarly, we do the same for validation steps on **Line 57**.

Finally, `NUM_CLASSES` defines the total number of classes in the dataset which we set to be the length of the `CLASSES` dictionary plus one for the background class.

Just as we have a *training* configuration, we also have an *inference/prediction* configuration as well:

```
62  class LesionBoundaryInferenceConfig(LesionBoundaryConfig):
63      # set the number of GPUs and images per GPU (which may be
64      # different values than the ones used for training)
65      GPU_COUNT = 1
66      IMAGES_PER_GPU = 1
67
68      # set the minimum detection confidence (used to prune out false
69      # positive detections)
70      DETECTION_MIN_CONFIDENCE = 0.9
```

Take note that our inference configuration is *not* a subclass of the Mask R-CNN `Config` class but is rather a subclass of our `LesionBoundaryConfig` used for training. We perform this subclassing out of convenience, ensuring we do not have to redefine variables that have already been defined before.

That said, I have included **Lines 65 and 66** demonstrating how you can override any variables — here you can adjust your `GPU_COUNT` and `IMAGES_PER_GPU` as you see fit.

I'd also like to callout **Line 70** where we define the `DETECTION_MIN_CONFIDENCE` which controls the minimum probability required for our Mask R-CNN to mark the given region as a "detection".

Here we set the minimum required confidence to 90% to help prune out false-positive detections. A value of 90% works well for the skin lesions dataset but you may way to adjust this value for your own projects.

When you train a Mask R-CNN on your own custom dataset I actually recommend setting `DETECTION_MIN_CONFIDENCE` to a small value, such as `0.5`. The primary reason is that you want to ensure that your network is actually learning *something*. If your minimum probability is too high then you may be unable to visually validate your network is learning. Start with a low detection confidence, validate your network is learning, and then increase it.

Our next class is responsible for actually managing the lesion dataset, including loading both images and their corresponding masks from disk:

```
71 class LesionBoundaryDataset(utils.Dataset):
72     def __init__(self, imagePaths, classNames, width=1024):
73         # call the parent constructor
74         super().__init__(self)
75
76         # store the image paths and class names along with the width
77         # we'll resize images to
78         self.imagePaths = imagePaths
79         self.classNames = classNames
80         self.width = width
```

Here we define our `LesionBoundaryDataset` which is a subclass of the `Dataset` class in the `mrcnn` module. Defining a dataset class is a *requirement* when working with the Matterport implementation of the Mask R-CNN. The dataset class defines the logic to load an image and load a mask.

Our constructor on **Line 73** requires two arguments followed by a third optional one:

1. `imagePaths`: Paths to our input images.
2. `classNames`: Our `CLASSES` dictionary.
3. `width`: The width that we'll be resizing images to prior to training. Providing a pre-defined width is extremely helpful when working with images with large spatial dimensions. Images that are too large can cause your GPU to run out of memory.

The following function can be used to initiate loading either our training or validation images:

```
83     def load_lesions(self, idxs):
84         # loop over all class names and add each to the 'lesion'
85         # dataset
86         for (classID, label) in self.classNames.items():
87             self.add_class("lesion", classID, label)
88
89         # loop over the image path indexes
90         for i in idxs:
91             # extract the image filename to serve as the unique
92             # image ID
93             imagePath = self.imagePaths[i]
94             filename = imagePath.split(os.path.sep)[-1]
95
96             # add the image to the dataset
97             self.add_image("lesion", image_id=filename,
98                 path=imagePath)
```

The `load_lesions` function accepts a single argument, the `idxs` of either the training or validation images (i.e., either `trainIdxs` or `valIdxs`).

Lines 86 and 87 loop over all class unique integer ID and human readable name pairs and adds them to the `lesion` dataset via the `add_class` method.

We then start looping over each of the `idxs` on **Line 90**. For each index `i`, we:

- Grab the corresponding `imagePath`
- Derive the filename from the image path
- Call `add_image`, adding the image to the `lesion` dataset and supplying the unique `image_id` and the path to the image itself.

The `Dataset` class of Mask R-CNN requires that we define two functions: `load_image` and `load_mask`. Both of these functions encapsulate all logic needed to load an input image from disk along with its corresponding mask.

Let's start by examining the `load_image` function:

```
100     def load_image(self, imageID):
101         # grab the image path, load it, and convert it from BGR to
102         # RGB color channel ordering
103         p = self.image_info[imageID]["path"]
104         image = cv2.imread(p)
105         image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
106 
107         # resize the image, preserving the aspect ratio
108         image = imutils.resize(image, width=self.width)
109 
110         # return the image
111         return image
```

The `load_image` method accepts a single argument which is our unique `imageID`. We can lookup all information passed into `add_image` (back on **Line 97 and 98** in the previous code block) by passing the `imageID` into the `image_info` dictionary maintained by the `Dataset` class.

Line 103 extracts the path to the input image which is then read from disk and converted from BGR to RGB channel ordering on **Lines 104 and 105**.

OpenCV stores images in BGR ordering; however, the Mask R-CNN library assumes RGB ordering, hence the channel reordering. **Line 108** then resizes our image to a fixed width.

Similarly, let's examine our `load_mask` method:

```
113     def load_mask(self, imageID):
114         # grab the image info and derive the full annotation path
115         # file path
116         info = self.image_info[imageID]
117         filename = info["id"].split(".")[0]
118         annotPath = os.path.sep.join([MASKS_PATH,
119             "{}_segmentation.png".format(filename)])
```

Just like `load_image`, the `load_mask` function requires a single argument — the unique image ID.

Our goal in this code block is to (1) take the image filename and (2) derive the corresponding mask file path from it. Since we know all input images have the format `isic_<7-digit unique ID>.jpg` and all masks have the format `isic_<7-digit unique ID>_segmentation.png`, we simply need to extract the first portion of the image filename and then append `_segmentation.png` to it to derive the mask filename (**Lines 116-119**).

We can then create the full annotation path by combining the `MASKS_PATH` root with the `filename`.

The next step is to load the actual mask from disk and extract all instances:

```
121         # load the annotation mask and resize it, *making sure* to
122         # use nearest neighbor interpolation
123         annotMask = cv2.imread(annotPath)
124         annotMask = cv2.split(annotMask)[0]
```

```
125             annotMask = imutils.resize(annotMask, width=self.width,
126                 inter=cv2.INTER_NEAREST)
127             annotMask[annotMask > 0] = 1
128
129             # determine the number of unique class labels in the mask
130             classIDs = np.unique(annotMask)
131
132             # the class ID with value '0' is actually the background
133             # which we should ignore and remove from the unique set of
134             # class identifiers
135             classIDs = np.delete(classIDs, [0])
```

Line 123 loads our input image from disk. OpenCV treats each input image it reads as a 3-channel RGB image (but with BGR ordering). Since we know our mask is a grayscale image (implying all R, G, and B channels have the same value), we can split the image into channels and discard all but the first one (**Line 124**).

The annotation mask is then resized to have the same width as our input image (**Lines 125 and 126**), taking care to apply ***nearest neighbor interpolation*** for reasons described in Section 3.1.1 — if you cannot recall why nearest neighbor interpolation is used, stop now and reread Section 3.1.1.

The ISIC 2018 masks have two values: `0` for background and `255` for foreground. However, our `CLASSES` requires us to have a value of `1` for all lesions, hence we threshold the `annotMask`, setting all values greater than zero to `1`, implying that all pixels with a value of `1` must be a lesion mask.

Line 130 computes the set of unique `classIDs` in the `annotMask` which should trivially be `0` (background) and `1` (foreground). Since the background class is unneeded we need to explicitly delete it from the `classIDs` (**Line 135**).

If you're attempting to train a Mask R-CNN with more than one class you'll need to modify the above code block, in particular **Line 127** — take care to ensure you can map each and every object in an image to its corresponding class ID.

The final step in the `load_mask` method is to create a mask for *every object* in the annotation mask:

```
137             # allocate memory for our [height, width, num_instances]
138             # array where each "instance" effectively has its own
139             # "channel" -- since there is only one lesion per image we
140             # know the number of instances is equal to 1
141             masks = np.zeros((annotMask.shape[0], annotMask.shape[1], 1),
142                 dtype="uint8")
```

On **Lines 141 and 142** we allocate memory for a `masks` array. The `masks` array should have the same width and height as the annotation mask (the first two dimensions).

The third dimension is the number of objects in the `annotMask`. For the ISIC 2018 skin lesion boundary detection challenge, the number of objects per image is *always* one. However, for your own dataset, you may have more than one object so you'll want to adjust this value accordingly.

Now that `masks` array is defined we can loop over all `classIDs`, which should again trivially be an array with only the value `1` in it:

```
144             # loop over the class IDs
145             for (i, classID) in enumerate(classIDs):
```

```
146                 # construct a mask for *only* the current label
147                 classMask = np.zeros(annotMask.shape, dtype="uint8")
148                 classMask[annotMask == classID] = 1
149
150                 # store the class mask in the masks array
151                 masks[:, :, i] = classMask
152
153             # return the mask array and class IDs
154             return (masks.astype("bool"), classIDs.astype("int32"))
```

For each of our unique `classIDs` we allocate memory for a `classMask` — this `mask` will be *binary*, having two values: `0` and `1`.

A value of `0` indicates that a pixel is part of the background. A value of `1` indicates that the pixel is foreground *and* belongs to the current `classID`.

Line 148 finds all (x,y)-coordinates in the `annotMask` with value `classID` and then sets all corresponding (x,y)-coordinates in `classMask` to one. We then store `classMask` in `masks` on **Line 151**.

Our `load_mask` function is *required* to return a 2-tuple on **Line 154**.

The first entry in the tuple is the `masks` array which *must* be returned as type boolean. Keep in mind that our `masks` array is required to have the shape `(width, height, num_instances)`.

The second entry in the tuple *must* have a length of `num_instances` as it maps each entry in `masks` to its corresponding classID. Since the ISIC 2018 dataset (1) consists of just one class label and (2) only one object per image, we can simply return the `classIDs` array.

Before we move on to the rest of the training script (which is *far* easier to comprehend and follow than all previous code blocks), I will say that the `load_mask` function will give you the biggest headaches when attempting to train your own Mask R-CNN.

I'll be providing my suggestions and best practices (including code) when defining both your `load_image` and `load_mask` function later in this chapter, but do keep in mind that training a Mask R-CNN is *less* about the algorithm and *more* about the data preparation.

You'll want to take ***excruciating care*** that `load_image` and your `load_mask` functions are defined correctly, otherwise you'll be left scratching your head looking at confusing error messages or nonsensical training results.

Again, I want to reiterate, these functions are the *lifeblood* of training your Mask R-CNN. *Do not* attempt to train your Mask R-CNN until you are positively certain they are behaving correctly *and* as you intended.

All that said, let's get into the code that can be used to train our Mask R-CNN and make predictions:

```
156 if __name__ == "__main__":
157     # construct the argument parser and parse the arguments
158     ap = argparse.ArgumentParser()
159     ap.add_argument("-m", "--mode", required=True,
160         help="either 'train', 'predict', or 'investigate'")
161     ap.add_argument("-w", "--weights",
162         help="optional path to pretrained weights")
163     ap.add_argument("-i", "--image",
164         help="optional path to input image to segment")
165     args = vars(ap.parse_args())
```

Line 156 checks to see if we are actually executing our script (i.e., typing `python lesions.py ...` in our command line).

Our script requires a single command line argument along with two optional ones:

1. `--mode`: Either `train`, `predict`, or `investigate`. The `train` mode will be responsible for actually training our Mask R-CNN while `predict` mode will be used exclusively for making predictions on input images. We can use the `investigate` mode for debugging and visually validating our dataset prior to training.
2. `--weights`: Path to our trained model weights during `predict` mode.
3. `--image`: Path to our input image during `predict` mode.

Let's go ahead and explore the `train` mode:

```
167     # check to see if we are training the Mask R-CNN
168     if args["mode"] == "train":
169         # load the training dataset
170         trainDataset = LesionBoundaryDataset(IMAGE_PATHS, CLASS_NAMES)
171         trainDataset.load_lesions(trainIdxs)
172         trainDataset.prepare()
173
174         # load the validation dataset
175         valDataset = LesionBoundaryDataset(IMAGE_PATHS, CLASS_NAMES)
176         valDataset.load_lesions(valIdxs)
177         valDataset.prepare()
178
179         # initialize the training configuration
180         config = LesionBoundaryConfig()
181         config.display()
```

Lines 170-172 constructs our `trainDataset`. Notice how we instantiate the `LesionBoundaryDataset` by supplying the `IMAGE_PATHS` and `CLASS_NAMES`.

From there, we pass the `trainIdxs` into `load_lesions` to instruct the `LesionBoundaryDataset` to only load the training images. A call to `.prepare` is the final step in preparing the training dataset.

Similarly, we perform the same set of actions on **Lines 175-177**, this time supplying the `valIdxs` to create the `valDataset`.

Line 180 initializes our training configuration. A call to `.display()` prints all configuration options and their corresponding values to our terminal — this information can be very helpful when debugging and tuning hyperparameters.

As mentioned earlier in this section, we will be applying data augmentation when training to both (1) reduce overfitting and (2) increasing the ability of our model to generalize:

```
183         # initialize the image augmentation process
184         aug = iaa.SomeOf((0, 2), [
185             iaa.Fliplr(0.5),
186             iaa.Flipud(0.5),
187             iaa.Affine(rotate=(-10, 10))
188         ])
```

Here we are defining our image augmentation rules. We'll start by picking `SomeOf`, either zero, one or two, of the following operations:

- Flipping horizontally with probability 0.5 (**Line 185**)
- Flipping vertically with probability 0.5 (**Line 186**).
- Randomly rotating between -10 and 10 degrees (**Line 187**).

When training the aug variable will be called and will apply our augmentation rules.

Finally, we are now ready to train our Mask R-CNN:

```
190         # initialize the model and load the COCO weights so we can
191         # perform fine-tuning
192         model = modellib.MaskRCNN(mode="training", config=config,
193             model_dir=LOGS_AND_MODEL_DIR)
194         model.load_weights(COCO_PATH, by_name=True,
195             exclude=["mrcnn_class_logits", "mrcnn_bbox_fc",
196                 "mrcnn_bbox", "mrcnn_mask"])
197
198         # train *just* the layer heads
199         model.train(trainDataset, valDataset, epochs=20,
200             layers="heads", learning_rate=config.LEARNING_RATE,
201             augmentation=aug)
202
203         # unfreeze the body of the network and train *all* layers
204         model.train(trainDataset, valDataset, epochs=40,
205             layers="all", learning_rate=config.LEARNING_RATE / 10,
206             augmentation=aug)
```

Lines 192 and 193 instantiates our Mask R-CNN object. Notice how (1) we are explicitly telling the model we are in `training` mode and (2) we pass in our training `config`.

Lines 194-196 load our COCO weights from disk. We'll be fine-tuning this model for instance segmentation on our lesions dataset.

As we learned from Chapter 5 in the *Practitioner Bundle*, we typically fine-tune our newly initialized layer heads *before* training the body of the network.

Lines 199-201 freeze the weights of the body and train *just* the layer heads. Here we supply both our `trainDataset` and `valDataset`, allowing the network to train for 20 epochs. We use the default learning rate of 0.001.

After the 20th epoch we then unfreeze the body and allow the *entire* network to train for an additional 20 epochs (for a total of 40 epochs). The learning rate is lowered by a factor of ten to help reduce overfitting.

Once our model is trained we can move on to the "predict" mode:

```
208     # check to see if we are predicting using a trained Mask R-CNN
209     elif args["mode"] == "predict":
210         # initialize the inference configuration
211         config = LesionBoundaryInferenceConfig()
212
213         # initialize the Mask R-CNN model for inference
214         model = modellib.MaskRCNN(mode="inference", config=config,
215             model_dir=LOGS_AND_MODEL_DIR)
216
217         # load our trained Mask R-CNN
218         weights = args["weights"] if args["weights"] \
219             else model.find_last()
220         model.load_weights(weights, by_name=True)
```

Line 211 initializes our *inference* configuration. We then initialize our Mask R-CNN in `inference` mode, supplying the inference configuration as well (**Lines 214 and 215**).

Lines 218-220 load the model weights from disk. We can either *explicitly* supply the path to the model weights via the `--weights` command line argument or we can use the `.find_last()` utility function of the `model` which examines our model snapshot directory and finds the latest model snapshot.

The next step is to prepare our image for prediction:

```
222         # load the input image, convert it from BGR to RGB channel
223         # ordering, and resize the image
224         image = cv2.imread(args["image"])
225         image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
226         image = imutils.resize(image, width=1024)
227
228         # perform a forward pass of the network to obtain the results
229         r = model.detect([image], verbose=1)[0]
```

We start by loading our input image from disk, and just like our `load_image` function, we reorder the channels from BGR to RGB ordering as well as resize the image to have a fixed with of 1024 pixels.

To obtain the detections and masks from our Mask R-CNN, we simply call the `model.detect` function. The returned value `r` is a dictionary containing four keys: `scores`, `masks`, `class_ids`, and `rois`.

Let's examine how we can use the keys to obtain the masks for each lesion:

```
231         # loop over of the detected object's bounding boxes and
232         # masks, drawing each as we go along
233         for i in range(0, r["rois"].shape[0]):
234             mask = r["masks"][:, :, i]
235             image = visualize.apply_mask(image, mask,
236                 (1.0, 0.0, 0.0), alpha=0.5)
237             image = visualize.draw_box(image, r["rois"][i],
238                 (1.0, 0.0, 0.0))
```

We start by looping over each of the detected bounding boxes (`rois`). **Line 234** extracts the actual `mask` for the current object.

If you examine the shape of the `mask` you will notice it has the same dimensions as the `image`, and furthermore, the mask will have two binary values:

1. `True`: The *foreground* region.
2. `False`: The *background* region.

Lines 235-236 call the `apply_mask` of the `visualize` submodule of Mask R-CNN. Under the hood, `apply_mask` is applying alpha blending to overlay the `mask` on our `image`. **Lines 237 and 238** draw our bounding box on the `image`.

At this point we are going to use OpenCV functions rather than the Mask R-CNN utility functions to draw on our `image`:

```
240         # convert the image back to BGR so we can use OpenCV's
241         # drawing functions
242         image = cv2.cvtColor(image, cv2.COLOR_RGB2BGR)
243
244         # loop over the predicted scores and class labels
245         for i in range(0, len(r["scores"])):
```

```
246                 # extract the bounding box information, class ID, label,
247                 # and predicted probability from the results
248                 (startY, startX, endY, end) = r["rois"][i]
249                 classID = r["class_ids"][i]
250                 label = CLASS_NAMES[classID]
251                 score = r["scores"][i]
252
253                 # draw the class label and score on the image
254                 text = "{}: {:.4f}".format(label, score)
255                 y = startY - 10 if startY - 10 > 10 else startY + 10
256                 cv2.putText(image, text, (startX, y),
257                     cv2.FONT_HERSHEY_SIMPLEX, 0.8, (0, 255, 0), 2)
```

Since we'll be using the OpenCV drawing functions, we need to convert our image back to BGR channel ordering (**Line 242**). On **Line 245** we loop over all predicted scores (i.e., probabilities).

It's important to note we don't need to explicitly threshold the scores to filter out weak detections here — that action is already performed for us via the `DETECTION_MIN_CONFIDENCE` value we suppled to the lesion boundary inference configuration.

For each score we extract the bounding box coordinates (**Line 248**) along with the `classID` and `label` (**Lines 249 and 250**). Given our bounding box coordinates, label, and score, we draw them on the `image` (**Lines 254-257**).

At last, we can display our output image to our screen:

```
259             # resize the image so it more easily fits on our screen
260             image = imutils.resize(image, width=512)
261
262             # show the output image
263             cv2.imshow("Output", image)
264             cv2.waitKey(0)
```

Line 260 resizes our image so that it can easily fit onto our screen while **Lines 263 and 264** actually show the image.

The final mode we are going to look at is used for investigating and debugging your dataset, images, and masks *before* you even start training. I provide more insight into the reasoning behind such a mode existing in Section 3.6 below, so for now, let's look at just the code:

```
273     # check to see if we are investigating our images and masks
274     elif args["mode"] == "investigate":
275         # load the training dataset
276         trainDataset = LesionBoundaryDataset(IMAGE_PATHS, CLASS_NAMES)
277         trainDataset.load_lesions(trainIdxs)
278         trainDataset.prepare()
279
280         # load the 0-th training image and corresponding masks and
281         # class IDs in the masks
282         image = trainDataset.load_image(0)
283         (masks, classIDs) = trainDataset.load_mask(0)
284
285         # show the image spatial dimensions which is HxWxC
286         print("[INFO] image shape: {}".format(image.shape))
287
288         # show the masks shape which should have the same width and
```

```
289         # height of the images but the third dimension should be
290         # equal to the total number of instances in the image itself
291         print("[INFO] masks shape: {}".format(masks.shape))
292
293         # show the length of the class IDs list along with the values
294         # inside the list -- the length of the list should be equal
295         # to the number of instances dimension in the 'masks' array
296         print("[INFO] class IDs length: {}".format(len(classIDs)))
297         print("[INFO] class IDs: {}".format(classIDs))
```

Lines 269-271 load our training dataset, just as if we were going to train our network. However, instead of training, we load the 0-th image and corresponding mask from the dataset.

Line 279 displays the spatial dimensions of the `image`. The actual spatial dimensions are *height* $\times$ *width* $\times$ *channels*.

Make sure these values are what you expect. For example, if you are resizing your images to have a fixed width or height, make sure that particular dimension has been properly resized. Similarly, if you're working with RGB images, make sure the number of channels is three.

Line 284 displays the dimensions of the `mask` volume which will have the shape *height* $\times$ *width* $\times$ *num_instances*. The `mask` volume should have the *same* width and height as the `image` volume; however, the third dimension should be equal to the total number of objects/instances in the image itself.

We then investigate the `classIDs` array on **Lines 289 and 290**. The length of the `classIDs` array should be equal to the total number of instances in the `masks` array (i.e., the third dimension).

You should also verify that the `classIDs` array contains only the unique label IDs for the objects in `masks` — the array should *not* contain a value of `0` as zero is reserved for the background.

The final step I recommend when investigating your dataset is to visually validate that the Mask R-CNN "thinks" your dataset is constructed properly.

To accomplish such as task we are going to loop over a randomly sampled set of indexes, loading the image and corresponding mask, and then use the `display_top_masks` helper utility to actually visualize the objects on our screen:

```
292         # determine a sample of training image indexes and loop over
293         # them
294         for i in np.random.choice(trainDataset.image_ids, 3):
295             # load the image and masks for the sampled image
296             print("[INFO] investigating image index: {}".format(i))
297             image = trainDataset.load_image(i)
298             (masks, classIDs) = trainDataset.load_mask(i)
299
300             # visualize the masks for the current image
301             visualize.display_top_masks(image, masks, classIDs,
302                 trainDataset.class_names)
```

The output of running the above code block can be seen in Figure 3.6. Here you can see that the Mask R-CNN framework and dataset utilities have been able to correctly read our training data — our input image is displayed to the screen and the "lesion" region is correctly masked.

I recommend you *always* perform such debugging *prior* to training your Mask R-CNN as it will save you significantly headaches down the line. I provide more tips and suggestions to successfully train Mask R-CNNs on your own dataset in Section 3.6, but for now, let's move on to training the actual network.

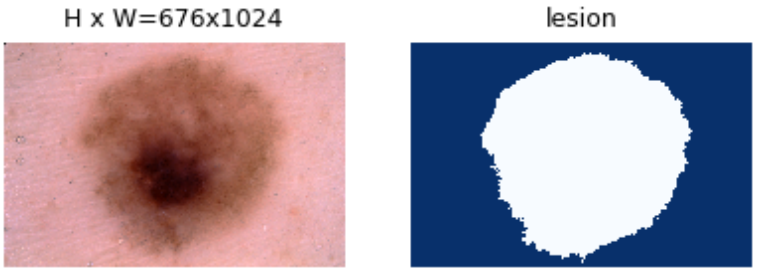


Figure 3.6: An example output of running `visualize.display_top_masks`. On the *left* is our input image. On the *right* is our corresponding mask. You should use the `investigate` mode to visually validate that the Keras + Mask R-CNN framework is properly parsing your dataset.

3.4.3 Training Your Mask R-CNN

That was quite a lot of code to review! Let's see how we can actually train our Mask R-CNN on the ISIC 2018 dataset. Open up a terminal and execute the following command:

```
$ python lesions.py --mode train
...
Starting at epoch 0. LR=0.001
Epoch 1/20
2075/2075 - 1234s 595ms/step - loss: 1.1190 - val_loss: 0.8835
Epoch 2/20
2075/2075 - 1192s 575ms/step - loss: 0.8643 - val_loss: 0.8789
Epoch 3/20
2075/2075 - 1176s 567ms/step - loss: 0.8274 - val_loss: 0.7446
...
```

Here you can see that our Mask R-CNN is starting to train. Each epoch is taking approximately 19 minutes on my Titan X GPU. Loss steadily decreases throughout training the layer heads.

Then, at the end of epoch 20, we unfreeze all layers and allow the *entire* network to be trained:

```
Starting at epoch 20. LR=0.0001
Epoch 21/40
2075/2075 - 1757s 847ms/step - loss: 0.4888 - val_loss: 0.5687
Epoch 22/40
2075/2075 - 1757s 847ms/step - loss: 0.4409 - val_loss: 0.6123
...
Epoch 39/40
2075/2075 - 1770s 853ms/step - loss: 0.2854 - val_loss: 0.5199
Epoch 40/40
2075/2075 - 1867s 900ms/step - loss: 0.2739 - val_loss: 0.5220
```

Notice that now we are training the entire network that epochs are now taking approximately 30 minutes on my Titan X. Again, both training and validation loss continue to drop but at the end of the training process we see loss is starting to saturate at 0.5, indicating that we likely cannot reduce loss further without additional hyperparameter adjustment.

3.5 Mask Predictions with Your Mask R-CNN

Now that our Mask R-CNN is trained, let's see how we can make predictions with it. Open up your terminal and execute the following command:

```
$ python lesions.py --mode predict \
    --image isic2018/ISIC2018_Task1-2_Training_Input/ISIC_0000000.jpg
```

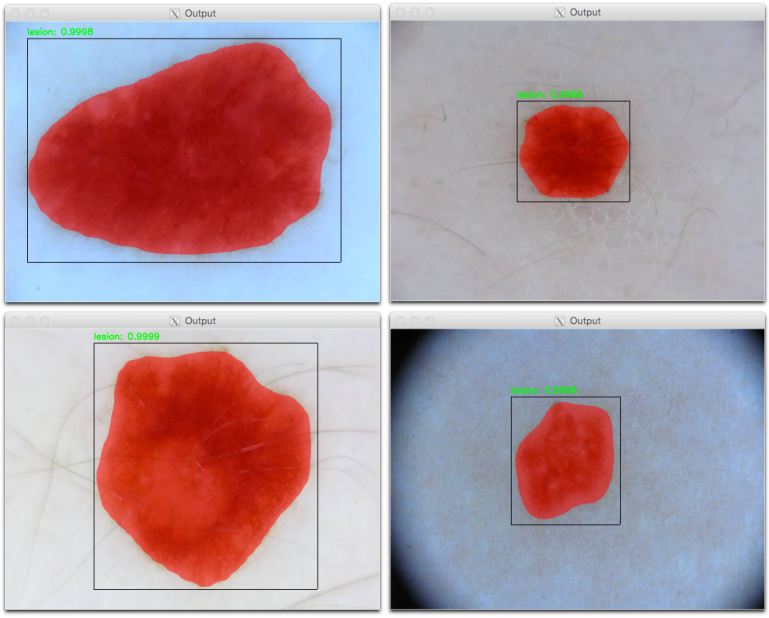


Figure 3.7: Sample predictions from our Mask R-CNN applied to skin lesion boundary detection. Notice how accurate our pixel-wise masks are.

I have included a montage of example mask predictions in Figure 3.7. Notice how our Mask R-CNN is doing an *extremely* good job at predicting the pixel-wise mask of the lesion.

Such a model could easily be used in a skin cancer prediction pipeline where the first step is detecting and extracting the region ROI. Further downstream steps of the pipeline could include classifying and identifying the skin lesion type (i.e., Task #3 of the ISIC 2018 challenge).

3.6 Tips When Training Your Own Mask R-CNN

Inevitably, you *will* run into problems when training your own Mask R-CNN, similar to how you likely ran into issues when training your first Faster R-CNN or SSD in Chapter 15 and 17 [REF] of the *ImageNet Bundle*.

Keep in mind that running into problems is the *norm* when working with state-of-the-art computer vision and deep learning libraries — if such computer vision solutions existed off the shelf it would be highly unlikely you would be reading this book in the first place.

In this section I'll detail my suggestions when training your own Mask R-CNN and how to overcome errors I personally encountered (and you likely will as well).

3.6.1 Training Never Starts

When I first started working with the `mrcnn` package I noticed a strange, intermittent behavior where my training script would get "stuck" at the start of Epoch #1 but never actually *start* training:

```
$ python lesions.py --mode train
...
Starting at epoch 0. LR=0.001
Epoch 1/20
...training never starts...
```

In fact, this behavior *never* occurred when working with the pills dataset in the case study in the following chapter, but *did* happen intermittently with the skin lesions dataset.

Instead of throwing my hands in the air and giving up, I first sought out the Matterport Mask R-CNN GitHub Issues page (http://pyimg.co/dr978). I then used the "search" feature to look for keywords related to "training hanging" or "training not starting". Through my short 2-3 minute research I came across a handful of separate threads that detailed the issue.

While the actual cause of the problem was never properly determined, the issue could be resolved by opening up the `Mask_RCNN/mrcnn/model.py` file and finding the `self.keras_model.fit_generator` call:

```
2365        self.keras_model.fit_generator(
2366            train_generator,
2367            initial_epoch=self.epoch,
2368            epochs=epochs,
2369            steps_per_epoch=self.config.STEPS_PER_EPOCH,
2370            callbacks=callbacks,
2371            validation_data=val_generator,
2372            validation_steps=self.config.VALIDATION_STEPS,
2373            max_queue_size=100,
2374            workers=workers,
2375            use_multiprocessing=True,
```

If you are on the same `mrcnn` branch as me, this code block should start around **Line 2364**. You'll want to then set `use_multiprocessing=False` — this fix allowed my Mask R-CNN to start training. The GitHub Issues threads I referenced above also said that setting `workers=0` was also necessary but I did not need to update the `workers` value.

The important takeaway here is that *you need to do your research* when you encounter an error.

There will ***absolutely*** be errors you encounter — that's part of the challenge in doing state-of-the-art computer vision and deep learning work. It is *your job* to do the necessary investigative work in trying to troubleshoot and resolve the problem. It may seem like a pain but this type of research and investigative work is a *necessary* and *critical* skill for you to master.

Keep at it, don't get discouraged, and do the work — it will all work out in the end.

3.6.2 Focus On Your Image and Mask Loading Functions

I cannot stress enough that you need to debug, re-debug, and re-re-debug your `load_image` and `load_mask` functions. The `load_mask` function in particular will be the one that gives you the most headaches.

Training a Mask R-CNN is *less* about the actual `mrcnn` library and *more* about the image and mask loading utilities. You need to double, triple, and quadruple check these functions *before* you even start training. **Don't rush into training, doing so will only cause you problems later!**

When implementing your own `load_image` and `load_mask` functions, make sure you refer to my `investigate` mode in the `lesions.py` file. Make sure you're calling both `load_image` and `load_mask` for a given set of images.

From there, you'll want to examine the shape and values of the returned NumPy arrays. Ask yourself:

1. Are my input image and mask the same spatial dimensions?
2. Have I remembered to convert from RGB to BGR channel ordering?
3. Are there multiple objects in each image and mask?
4. Can the multiple objects be *different* classes?
5. Have I defined my `masks` array with the proper number of instance as the final dimension?
6. How am I "finding" each individual object in the mask? Array indexing? A special lookup key that is provided with the dataset?
7. Have I ensured that the final dimensions of the `masks` array is the same length as the class identifiers I am returning from `load_mask`?

Use this set of questions and checklist as a starting point and then *add your own questions and items to the list* as you get more experienced with training Mask R-CNNs.

Once you've verified your `load_image` and `load_masks` functions are behaving as you expect, you'll want to visually validate your assumptions using the `display_top_masks` function (detailed in Section 3.4.2).

Visually validating that `mrcnn` can *correctly* parse out each object and corresponding mask should be an *absolute requirement* before trying to train your Mask R-CNN.

3.6.3 Refer to the Mask R-CNN Samples

Matterport, the maintainer of the Keras + Mask R-CNN implementation, has a number of super helpful samples of (1) loading your own custom data and (2) training a Mask R-CNN in their `samples` directory of the repository: http://pyimg.co/isgoe.

Inside the samples you'll find Jupyter Notebooks to guide you through various training examples. I personally found these examples *extremely* helpful and informative, *especially* when working with undocumented features.

I would *highly encourage* you to work through these examples as the educational experience is *well worth* the investment of your time.

3.7 Summary

In this chapter we discussed the differences between object detection, instance segmentation, and semantic segmentation. When performing object detection, our goal is to find all objects in an input image, label the objects, and provide the bounding box (x, y)-coordinates of each object.

Instance segmentation takes object detection a step further, providing not only the bounding box coordinates, *but also a pixel-wise mask of the object*, allowing us to segment the object from the image.

Semantic segmentation builds on instance segmentation but requires us to label *all pixels in an image*, thereby assigning a label to each pixel. Semantic segmentation is most commonly applied in "scene understanding" applications, such as self-driving cars.

Semantic segmentation algorithms are challenging to say the least, and furthermore, there most applications require only instance segmentation — we chose to study instance segmentation in this chapter.

To facilitate our understanding of instance segmentation, we reviewed the Mask R-CNN algorithm [12] which builds on the Faster R-CNN work by Girshick et al. [9] We then used the Keras Mask R-CNN implementation [13] to train our own custom instance segmentation model on the ISIC 2018 dataset [25, 26] — this model is capable of localizing skin lesion regions for cancer detection with high accuracy.

The ISIC 2018 dataset was pre-compiled for us though. How might we go about *labeling and annotating our own custom dataset*, followed by training a Mask R-CNN on our custom dataset? To answer that question, you'll need to refer to the next chapter.

4. Annotating Images and Training Mask R-CNN

In our previous chapter we learned how to train a Mask R-CNN to detect and segment skin cancer regions from an image, paving the way for a fully automatic computer vision system to detect, classify, and assess the risk of skin cancer on a patient.

However, it's important to note that the ISIC 2018 skin lesion boundary detection dataset [25, 26] was *pre-labeled* for us. The curators of the dataset had *already* provided us with the masks for each individual image.

But what if we wanted to work with our own *custom* dataset? What if we didn't *already* have the corresponding masks? How we would go about taking our input images and then annotate them such that we could (1) create the masks and then (2) train a Mask R-CNN.

In this chapter we'll explore the answer to these questions. I'll be showing you how to use image annotation tools to create masks for your own custom dataset.

I'll then not only *discuss* how to train a Mask R-CNN on top of your annotated data, but also *provide code* for you to complete the task. The code will serve as a template that you can modify and adjust as you see fit for your own project.

With that said, let's go ahead and get started!

4.1 Annotating Images for a Mask R-CNN

In the first part of this section I'll describe the example prescription pills image dataset that we'll be using. Our goal will be to correctly detect and extract each and every prescription pill from an input image. Being able to obtain a pixel-wise mask of a pill region is a critical first step in building a system to automatically recognize and identify prescription medication.

From there I'll provide suggestions for annotation tools that you can use for labeling and creating masks for your input images.

Finally, I'll provide an example of how I personally annotated all images in our pills dataset.

4.1.1 The Pills Dataset

Our pills image dataset actually a subset of the National Library of Medicine's (NLM) 2016 pill identification challenge [28].

Figure 4.1: A sample of the 41 image pills dataset, subset of the National Library of Medicine's (NLM) 2016 pill identification challenge [28]. Our goal is to train a Mask R-CNN network to predict a pixel-wise mask of every pill in an image.

Each year, over 3.3 million injuries and deaths happen each year to the incorrect prescription pill being taken. These mistakes happen both on both the patient and the doctor/pharmacist side.

As patients, it's not uncommon, especially among the elderly, to accidentally mix up their prescription medications and take the wrong one at the wrong time.

And similarly, while we like to think of doctors, nurses, and pharmacists as infallible, incapable of making mistakes, the truth is that sometimes a doctor may write the wrong prescription, thinking one thing, while the patient thinks they are receiving a different medication.

Pharmacists too make mistakes — just a couple months ago a pharmacist filled one of my prescriptions incorrectly. I was supposed to receive fifty pills and they incorrectly filled the prescription with only twenty-five.

Mistakes surrounding prescription medication can and do happen, resulting in *billions of dollars* in insurance claims, hospital bills, and of course, the incalculable value of a human life. Prescription pill identification is a research subject that I've studied for years and I'm incredibly happy to be sharing it with you in this chapter.

As for our pills dataset itself, our subset of the NLM dataset includes 41 images, some of which are displayed in Figure 4.1.

As we can see, applying simple image processing such as edge detection, thresholding, and contour extraction is not going to be viable — there is too much shadowing and not enough contrast between the background and foreground to obtain a reasonable segmentation of the pill.

Furthermore, since nearly 50% of all 30,000+ prescription pills in the United States are round and/or white, we need to devise a system that can segment pills in challenging, unknown environment conditions (i.e., pills on a kitchen/bathroom counter, in a hand, etc.).

The 41 images used in our dataset *do not* have corresponding masks, and thus to train a Mask

R-CNN, we need to *annotate* these images, both by labeling and creating the corresponding masks.

4.1.2 Annotation Tools

There are a number of image annotation tools that can be used for both object detection and image segmentation. Exactly which tool you should use is highly dependent on what environment you feel comfortable using.

There is an excellent Quora thread (http://pyimg.co/oon1j) that provides suggestions on various annotation tools across operating systems and web browsers.

In the context of this chapter, I'll be reviewing the tools that I see most people using, including myself.

Photoshop and Gimp

The first set of tools you may consider for annotating your own image dataset are the most obvious ones, including full-blown image manipulation software packages such as Photoshop and Gimp.

When using one of these tools your workflow would look something like:

- Load a single image from a dataset into the software
- Use "drawing" tools such as rectangles, circles, ellipses, or the polygon tool to draw out the mask
- Fill the mask with a specific color. Each label will have its own color.
- Save the mask to disk in a lossless image file format (such as PNG).

Such a workflow is totally viable for annotating an image dataset. The problem is that it can become extremely tedious to use such a tool — Photoshop and Gimp are not designed for image annotation.

Furthermore, such a tool compounds the element of human error. It's far too easy for you to forget which color corresponds to which class label.

Secondly, there is no way to impose a class label on the image/mask inside the software. In order to assign a class label you would need to define a special directory structure or maintain some sort of lookup text, JSON, YAML, etc. file.

RectLabel

A better solution would be to use a tool such as RectLabel (http://pyimg.co/nr37n) which enables you to draw bounding boxes, polygons, and cubic bezier splines.

The tool is designed with both object detection and segmentation, enabling you to export indexed color masks, making it fairly straightforward to integrate into your `load_mask` function. RectLabel also includes the ability for 1-click buttons to apply labels to ROIs.

The downside to RectLabel is that it's macOS only, so if you do not have a Mac, you'll need to try a different tool.

imglab

The imglab tool (http://pyimg.co/duem0) is part of the dlib library [29] and is actually used mainly for object detection. Using imglab you can draw a bounding box surrounding an object and then provide a label.

If all objects you are annotating are rectangular then imglab is a great option — you can annotate each individual image and imglab will generate an XML file with all annotation data. Given this annotation data you can update your `Dataset` subclass to parse the XML file and use OpenCV's `cv2.rectangle` function to draw the actual mask.

If your masks are *not* strictly rectangular then you will need to use a different tool.

LabelMe

LabelMe (http://pyimg.co/h31ee), inspired my MIT's tool of the same name, has become somewhat of a standard for image annotation, including object detection and instance segmentation.

The tool is fully open source, written in Python, and designed for use with the Qt graphical interface.

While the tool is fully operational and incredibly useful, I personally have found the interface a bit slow, "clunky", and in general less intuitive than some of the other tools in this section. Certainly give it a try, but I think you'll end up opting for a different tool.

LabelImg

LabelImg (http://pyimg.co/lqwwz) is essentially an offshoot of the LabelMe annotator.

LabelImg also relies on Python and Qt, although Windows and Linux users will be happy to know that binaries for the application exist — you do not need to explicitly build them like you would for LabelMe. Readers using macOS will still need to build the project from source, however.

Just like LabelMe, I find LabelImg a bit clunky and hard to use. I would still encourage you to give it a try though.

Labelbox

Perhaps one of my favorite image annotators is Labelbox (http://pyimg.co/fqk72). Unlike the other projects, which are open source, Labelbox is a SaaS (Software as a Service) application.

Both free, paid, and hosted plans exist for the product, but the general gist is that there is nothing to download and nothing to install — you simply point your browser to your data, either via uploading to the LabelBox server or running locally, and then annotate *directly* within your browsers.

With modern browsers behaving more like "operating systems" and less like "web browsers", I see very little reason why we should be confined to downloadable pieces of software for image annotation.

VGG Image Annotator (VIA)

My *personal favorite* image annotator is the VGG Image Annotator (VIA) (http://pyimg.co/d4ew1) [30] from the renowned Visual Geometry Group at the University of Oxford.

VIA is also an open source project but has the primary benefit of (1) running in your browser and (2) being *entirely* self-contained — all you need to do is download a HTML file and open it in your favorite web browser.

From there, the user interface guides you to loading your image dataset. You can then annotate your images for both object detection and instance segmentation.

Once you have finished annotating your images you can download a JSON or CSV file containing the segmentations. You can also save the project itself, ensuring that if you close your browser you can reload the project from where you left off and continue annotating.

4.1.3 Annotating the Pills Dataset

When annotating the pills dataset I choose to use the VIA tool — it's my personal favorite and includes a "circle" annotation tool that can easily be used to annotate our circular pills in the image dataset.

Figure 4.2 provides an illustration of myself using the VIA tool. Notice how I have used the "circle" tool to select the pill region. I then labeled the pill region as such.

After I finished annotating each of the 41 images, which tool approximately 30 minutes, I downloaded a JSON file of my dataset and named it `via_region_data.json`, the default name provided by the VIA tool.

Opening the JSON file in your favorite text editor, the contents would look similar to the following:

```
1 {
```

```
2       "0.jpg4422650":
3       {
4            "filename":"0.jpg",
5            "size":4422650,
6            "regions":
7            [{
8                 "shape_attributes":
9                 {
10                     "name": "circle",
11                     "cx": 2205,
12                     "cy": 1656,
13                     "r": 174
14                },
15                "region_attributes":
16                {
17                     "round": "1"
18                }
19           }],
20           "file_attributes": {}
21      }
22 }
```

Here we can see an example entry in the JSON file. **Line 2** defines the *unique ID* of the image which is a concatenation of both the image `filename` (**Line 4**) along with the file `size` (**Line 5**).

The `regions` list contains all *annotations* for the image. Each entry in the `regions` list contains two dictionaries: a `shape_attributes` and a `region_attributes`.

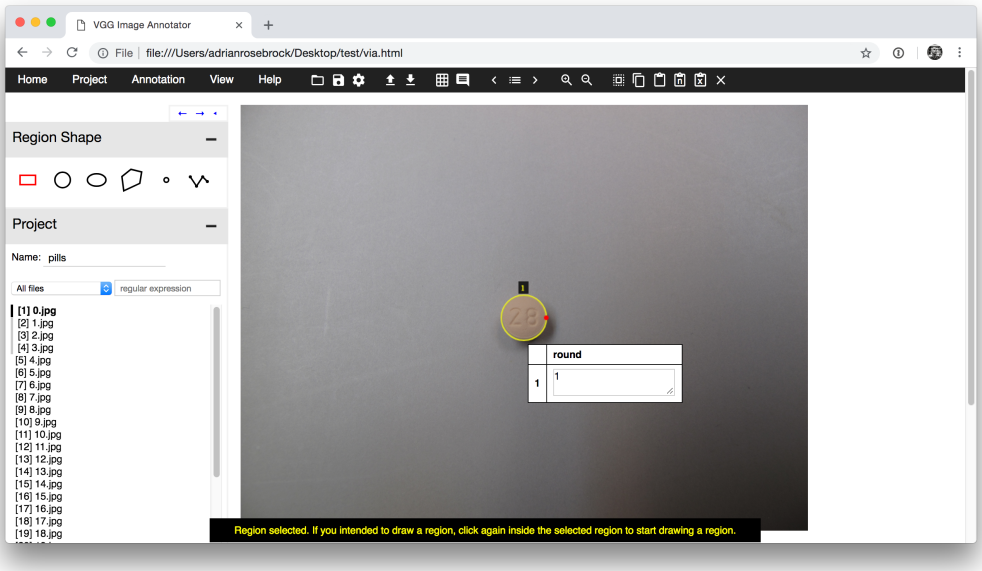


Figure 4.2: Using the VGG Image Annotator (VIA) to annotate our pills. Here we are drawing a *mask* for each pill and then providing the *label* as well.

The `shape_attributes` contains all information required to actually *draw* the mask.

Line 10 defines the `name` of the shape as a `circle` (other shapes exist in the VIA tool as well). To draw a circle we must know the center (x, y)-coordinates of the circle (**Lines 11 and 12**)

followed by the radius as well (**Line 13**). Using this information we'll be able to utilize OpenCV's drawing functions, specifically `cv2.circle`, to draw the mask inside the `load_mask` function.

Finally, the `region_attributes` contains the actual class label. Here, `round` is set equal to `1` to indicate that the class label for the shape attribute is "round".

In Section 4.2.2 I'll be demonstrating how we can parse our VIA-generated JSON file and use to construct our masks inside the `load_mask` function.

4.2 Training the Mask R-CNN to Segment Pills

In this section we'll start by exploring the directory structure of our project.

Next, we'll implement our Mask R-CNN training script, a near identical replica of our training script from Chapter 3 but with a few modifications. From there we'll train our Mask R-CNN for pill instance segmentation.

Finally, we'll review our results and apply our pill segmentation model to images *outside* the original training set.

4.2.1 Project Structure

Before we get started implementing the Mask R-CNN driver script for the pills dataset, let's first examine directory structure of our project:

```
|--- mask_rcnn
|    |--- pills/
|    |    |--- images
|    |    |--- via_region_data.json
|    |--- pills.py
|    |--- mask_rcnn_coco.h5
|    |--- mrcnn/
```

The `pills` directory contains our actual pills dataset described in Section 4.1.1. Inside the `pills` directory we have a subdirectory of `images` which are our actual image themselves. The `via_region_data.json` file contains the exported JSON mask annotations from the VIA tool we utilized in Section 4.1.3.

The `examples` directory contains example pill images that are *outside* our actual pills dataset. We'll be using these example images to visually validate that our Mask R-CNN is (1) working properly and (2) can generalize to images *outside* of our training and validation set.

The `pills.py` file is contains all Python code necessary to train our Mask R-CNN as well as make predictions with it. When training our Mask R-CNN, the framework will automatically populate the `pills_logs` directory with logs and model snapshots.

We then have the `pills_project.json` and `via.html` files. The `via.html` file is our actual VIA annotation tool.

The `pills_project.json` is the actual VIA project itself. We can load this file into VIA to restore our project and continue annotating if need be.

4.2.2 Implementing the Mask R-CNN Driver Script

Our `pills.py` script is *extremely* similar to our `lesions.py` script from Chapter 3 which is actually a *huge benefit* of using a framework such as the Matterport Keras Mask R-CNN implementation — it enables us to focus *less* on the code itself and more on actually training our network.

But as we know, the most crucial aspect of working with such a framework is ensuring our dataset has been properly parsed and loaded. I'll be spending less time explaining this script as

we've already reviewed such a training script in detail in our previous chapter; however, I will be calling out important differences along the way.

Let's go ahead and get started! Open up the `pills.py` file on your system and insert the following code:

```
1  # import the necessary packages
2  from mrcnn.config import Config
3  from mrcnn import model as modellib
4  from mrcnn import visualize
5  from mrcnn import utils
6  from imutils import paths
7  import numpy as np
8  import argparse
9  import imutils
10 import random
11 import json
12 import cv2
13 import os
```

Lines 2-13 handle importing our required Python packages — none of these packages are different from our previous training script in Chapter 3, except for the addition of `json` which will be used to parse our exported annotation data from the VIA tool.

Next, we define our `DATASET_PATH`, `IMAGES_PATH`, and `ANNOT_PATH`:

```
15 # initialize the dataset path, images path, and annotations file path
16 DATASET_PATH = os.path.abspath("pills")
17 IMAGES_PATH = os.path.sep.join([DATASET_PATH, "images"])
18 ANNOT_PATH = os.path.sep.join([DATASET_PATH, "via_region_data.json"])
19
20 # initialize the amount of data to use for training
21 TRAINING_SPLIT = 0.75
22
23 # grab all image paths, then randomly select indexes for both training
24 # and validation
25 IMAGE_PATHS = sorted(list(paths.list_images(IMAGES_PATH)))
26 idxs = list(range(0, len(IMAGE_PATHS)))
27 random.seed(42)
28 random.shuffle(idxs)
29 i = int(len(idxs) * TRAINING_SPLIT)
30 trainIdxs = idxs[:i]
31 valIdxs = idxs[i:]
```

Line 16 defines the root `DATASET_PATH` which is our `pills` directory. All images inside the `DATASET_PATH` reside in the `images` subdirectory. The `ANNOT_PATH` is the path to our exported JSON annotation data from the VIA tool.

Line 21 indicates that we'll be using 75% of our data for training and 25% for validation. The training and validation image path splits are constructed on **Lines 25-31**.

From there we can define our `CLASS_NAMES` dictionary:

```
33 # initialize the class names dictionary
34 CLASS_NAMES = {1: "round"}
```

```
35
36  # initialize the path to the Mask R-CNN pre-trained on COCO
37  COCO_PATH = "mask_rcnn_coco.h5"
38
39  # initialize the name of the directory where logs and output model
40  # snapshots will be stored
41  LOGS_AND_MODEL_DIR = "logs"
```

Our project has only a single class label, round, which is used to indicate a round prescription pill.

The `COCO_PATH` stores the path to our Mask R-CNN with ResNet backbone pre-trained on the COOC dataset [5]. We'll be fine-tuning this model later in the script.

We are now ready to define our `PillsConfig` for training:

```
43  class PillsConfig(Config):
44      # give the configuration a recognizable name
45      NAME = "pills"
46
47      # set the number of GPUs to use training along with the number of
48      # images per GPU (which may have to be tuned depending on how
49      # much memory your GPU has)
50      GPU_COUNT = 1
51      IMAGES_PER_GPU = 1
52
53      # set the number of steps per training epoch
54      STEPS_PER_EPOCH = len(trainIdxs) // (IMAGES_PER_GPU * GPU_COUNT)
55
56      # number of classes (+1 for the background)
57      NUM_CLASSES = len(CLASS_NAMES) + 1
```

We start by defining the `NAME` of our dataset as `pills` (**Line 45**).

We'll be training using a single GPU with a batch size of only one image per GPU (**Line 51**). On my 12GB Titan X GPU I can technically train with 2-4 images at a time; however, not all readers will have a GPU with as much memory so I left the value as one to ensure your an reproduce my results. Feel free to increase the value for faster training if you have a GPU with sufficient memory.

The number of steps per epoch is defined on **Line 54** while the total number of `NUM_CLASSES` is set to be the length of the `CLASS_NAMES` dictionary plus one for the background (**Line 57**).

Just as we have a pills configuration for training, we also need one for inference/prediction:

```
59  class PillsInferenceConfig(PillsConfig):
60      # set the number of GPUs and images per GPU (which may be
61      # different values than the ones used for training)
62      GPU_COUNT = 1
63      IMAGES_PER_GPU = 1
64
65      # set the minimum detection confidence (used to prune out false
66      # positive detections)
67      DETECTION_MIN_CONFIDENCE = 0.9
```

Take note of **Line 67** where we set the `DETECTION_MIN_CONFIDENCE` to be 90%. Any predictions with less than 90% confidence will be discarded during prediction.

Next comes our `PillsDataset` class, arguably the most important code in the entire `pills.py` script:

```
69 class PillsDataset(utils.Dataset):
70     def __init__(self, imagePaths, annotPath, classNames, width=1024):
71         # call the parent constructor
72         super().__init__(self)
73
74         # store the image paths and class names along with the width
75         # we'll resize images to
76         self.imagePaths = imagePaths
77         self.classNames = classNames
78         self.width = width
79
80         # load the annotation data
81         self.annots = self.load_annotation_data(annotPath)
```

Line 70 defines the constructor to our class which requires three parameters followed by a fourth optional one.

The `imagePaths` variable contains all image paths residing on disk. The `annotPath` is the path to our exported JSON mask annotations from the VIA tool.

We then have `classNames` which are the names of the labels in our dataset. Finally, `width` can be used to resize our images and masks to a fixed size prior to training.

We call our `load_annotation_data` helper method on **Line 81** to load and parse our actual annotations.

Speaking of `load_annotation_data`, let's go ahead and define that now:

```
83 def load_annotation_data(self, annotPath):
84         # load the contents of the annotation JSON file (created
85         # using the VIA tool) and initialize the annotations
86         # dictionary
87         annotations = json.loads(open(annotPath).read())
88         annots = {}
89
90         # loop over the file ID and annotations themselves (values)
91         for (fileID, data) in sorted(annotations.items()):
92             # store the data in the dictionary using the filename as
93             # the key
94             annots[data["filename"]] = data
95
96         # return the annotations dictionary
97         return annots
```

This method accepts a single argument which is assumed to be our path to the input JSON annotation file. **Line 87** loads the annotation data from disk and then initializes `annot`, a dictionary that will be used to map the filename (key) to the data itself (value).

We start looping over each of the unique file identifiers (filename + image file size) on **Line 91**. We then extract the filename from the `data` and store the `data` in the dictionary. **Line 97** returns the `annots` dictionary to the calling function.

The next function we'll define is `load_pills`:

```
99 def load_pills(self, idxs):
100         # loop over all class names and add each to the 'pills'
101         # dataset
```

```
102             for (classID, label) in self.classNames.items():
103                  self.add_class("pills", classID, label)
104
105             # loop over the image path indexes
106             for i in idxs:
107                  # extract the image filename to serve as the unique
108                  # image ID
109                  imagePath = self.imagePaths[i]
110                  filename = imagePath.split(os.path.sep)[-1]
111
112                  # load the image and resize it so we can determine its
113                  # width and height (unfortunately VIA does not embed
114                  # this information directly in the annotation file)
115                  image = cv2.imread(imagePath)
116                  (origH, origW) = image.shape[:2]
117                  image = imutils.resize(image, width=self.width)
118                  (newH, newW) = image.shape[:2]
119
120                  # add the image to the dataset
121                  self.add_image("pills", image_id=filename,
122                       width=newW, height=newH,
123                       orig_width=origW, orig_height=origH,
124                       path=imagePath)
```

The `load_pills` accepts a single argument, `idxs`, which will be either our training or validation indexes.

On **Lines 102 and 103** we loop over each of the unique integer class IDs and human readable labels in `classNames` and then add them to the `pills` dataset.

Line 106 starts a separate `for` loop, this one responsible for adding image data itself to the `pills` dataset. **Lines 109 and 110** derive the `filename` from the `imagePath`.

Lines 112-118 load our `image` from disk, determine the *original* spatial dimensions, resize the image, and then determine the *new* width and height. Unfortunately, the VIA annotation tool does not embed the spatial dimensions of the image into the JSON file, requiring us to explicitly load the image and determine the width and height.

For small datasets this code is fine, but if you were working with tens of thousands (or more) of images then you would want to consider creating a separate Python script to load the VIA JSON annotation data and add the image dimensions to it — trying to include such information in our training script introduces N I/O operations where N is the total number of images in our dataset, ideally something we would like to avoid.

Lines 121-124 add our image information to the `pills` dataset.

Our `load_image` function is identical to the `load_image` function from Chapter 3:

```
126 def load_image(self, imageID):
127             # grab the image path, load it, and convert it from BGR to
128             # RGB color channel ordering
129             p = self.image_info[imageID]["path"]
130             image = cv2.imread(p)
131             image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
132
133             # resize the image, preserving the aspect ratio
134             image = imutils.resize(image, width=self.width)
135
```

```
136             # return the image
137             return image
```

Here we extract image path, load the image, and reorder the channels from BGR to RGB ordering. The image is then resized to a pre-specified width and returned to the calling function.

Our `load_mask` function on the other hand is slightly different as we need to use the JSON annotation data to *draw* our actual mask rather than *load it from disk* like we did in the previous chapter:

```
139     def load_mask(self, imageID):
140         # grab the image info and then grab the annotation data for
141         # the current image based on the unique ID
142         info = self.image_info[imageID]
143         annot = self.annots[info["id"]]
144 
145         # allocate memory for our [height, width, num_instances] array
146         # where each "instance" effectively has its own "channel"
147         masks = np.zeros((info["height"], info["width"],
148             len(annot["regions"])), dtype="uint8")
149 
150         # loop over each of the annotated regions
151         for (i, region) in enumerate(annot["regions"]):
152             # allocate memory for the region mask
153             regionMask = np.zeros(masks.shape[:2], dtype="uint8")
154 
155             # grab the shape and region attributes
156             sa = region["shape_attributes"]
157             ra = region["region_attributes"]
158 
159             # scale the center (x, y)-coordinates and radius of the
160             # circle based on the dimensions of the resized image
161             ratio = info["width"] / float(info["orig_width"])
162             cX = int(sa["cx"] * ratio)
163             cY = int(sa["cy"] * ratio)
164             r = int(sa["r"] * ratio)
165 
166             # draw a circular mask for the region and store the mask
167             # in the masks array
168             cv2.circle(regionMask, (cX, cY), r, 1, -1)
169             masks[:, :, i] = regionMask
170 
171         # return the mask array and class IDs, which for this dataset
172         # is all 1's
173         return (masks.astype("bool"), np.ones((masks.shape[-1],),
174             dtype="int32"))
```

We start by looking up all image `info` on **Line 142** followed by grabbing all annotation data for the image on **Line 143**

We allocate memory for a `masks` array on **Line 147 and 148**, ensuring it has the same width and height as the corresponding image.

The third dimension is the number of instances in the object. All images in our dataset trivially have only one pill, but I've sent the third dimension to be the length of the `annot["regions"]` to demonstrate how you would dynamically set this value.

We then start looping over each of the annotated regions on **Line 151**.

Line 153 allocates memory for this *specific* `regionMask` (which will then be stored in `masks` after we draw on it).

Lines 156 and 157 grab our shape and region attributes, respectively. The shape attributes (`sa`) contain all information required to draw our circle mask.

Lines 161-164 scale the center (x, y)-coordinates and radius of the circle baed on the dimensions of the *resized* image.

Keep in mind that when we annotated our images we did so on the *full resolution* images. Since we are resizing our images to have a fixed width of 1024 pixels in the training script, we now need to scale our circle attributes as well.

Given our scaled center (x, y)-coordinates and radius, we can use the `cv2.circle` function to draw the actual mask. Supplying a value of -1 as the final parameter instructs OpenCV to "fill in" the circle instead of drawing just the outline. The `regionMask` is then stored in the `masks` array.

Our final step is to return the `masks` array and corresponding class IDs for each object in `masks`. Since we only have one class in our pills dataset, we'll return an array of ones that has the same length as the number of objects in the image.

Now that our configuration and dataset classes are defined, we can move on to training our Mask R-CNN on the pills dataset:

```
1  if __name__ == "__main__":
2      # construct the argument parser and parse the arguments
3      ap = argparse.ArgumentParser()
4      ap.add_argument("-m", "--mode", required=True,
5          help="either 'train', 'predict', or 'investigate'")
6      ap.add_argument("-w", "--weights",
7          help="optional path to pretrained weights")
8      ap.add_argument("-i", "--image",
9          help="optional path to input image to segment")
10     args = vars(ap.parse_args())
```

Lines 178-185 parse our command line arguments. Please refer to Section 3.4.2 of the previous chapter for a detailed review of each of the command line arguments.

Provided we are in the `train` mode, let's construct our training and validation dataset along with the training configuration object:

```
187 # check to see if we are training the Mask R-CNN
188     if args["mode"] == "train":
189         # load the training dataset
190         trainDataset = PillsDataset(IMAGE_PATHS, ANNOT_PATH,
191             CLASS_NAMES)
192         trainDataset.load_pills(trainIdxs)
193         trainDataset.prepare()
194 
195         # load the validation dataset
196         valDataset = PillsDataset(IMAGE_PATHS, ANNOT_PATH,
197             CLASS_NAMES)
198         valDataset.load_pills(valIdxs)
199         valDataset.prepare()
200 
201         # initialize the training configuration
202         config = PillsConfig()
203         config.display()
```

From there we can load our Mask R-CNN from disk and train it:

```
205         # initialize the model and load the COCO weights so we can
206         # perform fine-tuning
207         model = modellib.MaskRCNN(mode="training", config=config,
208             model_dir=LOGS_AND_MODEL_DIR)
209         model.load_weights(COCO_PATH, by_name=True,
210             exclude=["mrcnn_class_logits", "mrcnn_bbox_fc",
211                 "mrcnn_bbox", "mrcnn_mask"])
212
213         # train *just* the layer heads
214         model.train(trainDataset, valDataset, epochs=10,
215             layers="heads", learning_rate=config.LEARNING_RATE)
216
217         # unfreeze the body of the network and train *all* layers
218         model.train(trainDataset, valDataset, epochs=20,
219             layers="all", learning_rate=config.LEARNING_RATE / 10)
```

On **Lines 214 and 215** we train *just* the layer heads, freezing the body of the network. We perform this training procedure for a total of ten epochs.

From there, we unfreeze the body of the network and continue to train for another ten epochs (for a total of 20 epochs) on **Lines 218 and 219**. At the end of every epoch the Mask R-CNN framework snapshots our model weights to disk.

Next, we'll make a check to see if we are making predictions:

```
221     # check to see if we are predicting using a trained Mask R-CNN
222     elif args["mode"] == "predict":
223         # initialize the inference configuration
224         config = PillsInferenceConfig()
225
226         # initialize the Mask R-CNN model for inference
227         model = modellib.MaskRCNN(mode="inference", config=config,
228             model_dir=LOGS_AND_MODEL_DIR)
229
230         # load our trained Mask R-CNN
231         weights = args["weights"] if args["weights"] \
232             else model.find_last()
233         model.load_weights(weights, by_name=True)
234
235         # load the input image, convert it from BGR to RGB channel
236         # ordering, and resize the image
237         image = cv2.imread(args["image"])
238         image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
239         image = imutils.resize(image, width=1024)
```

Other than **Line 224** where we insatiate a `PillsInferneceConfig`, the code in the "predict" mode is identical to the previous chapter.

Lines 227 and 228 initialize the Mask R-CNN for inferences. **Lines 231-233** load the weights themselves for the Mask R-CNN.

From there we load our input `--image` from disk and preprocess it.

We are now ready to predict the masks and bounding boxes for any objects in the image:

```
241         # perform a forward pass of the network to obtain the results
242         r = model.detect([image], verbose=1)[0]
243
244         # loop over of the detected object's bounding boxes and
245         # masks, drawing each as we go along
246         for i in range(0, r["rois"].shape[0]):
247             mask = r["masks"][:, :, i]
248             image = visualize.apply_mask(image, mask,
249                 (1.0, 0.0, 0.0), alpha=0.5)
250             image = visualize.draw_box(image, r["rois"][i],
251                 (1.0, 0.0, 0.0))
```

A forward pass of the network is performed on **Line 242** yielding the actual prediction results.

Lines 246-251 loop over each of the detected bounding boxes and corresponding masks. Both the bounding box and mask are then drawn on the `image`.

From there we convert the image back to BGR channel ordering so we can use OpenCV's drawing functions:

```
253         # convert the image back to BGR so we can use OpenCV's
254         # drawing functions
255         image = cv2.cvtColor(image, cv2.COLOR_RGB2BGR)
256
257         # loop over the predicted scores and class labels
258         for i in range(0, len(r["scores"])):
259             # extract the bounding box information, class ID, label,
260             # and predicted probability from the results
261             (startY, startX, endY, end) = r["rois"][i]
262             classID = r["class_ids"][i]
263             label = CLASS_NAMES[classID]
264             score = r["scores"][i]
265
266             # draw the class label and score on the image
267             text = "{}: {:.4f}".format(label, score)
268             y = startY - 10 if startY - 10 > 10 else startY + 10
269             cv2.putText(image, text, (startX, y),
270                 cv2.FONT_HERSHEY_SIMPLEX, 0.8, (0, 255, 0), 2)
271
272         # resize the image so it more easily fits on our screen
273         image = imutils.resize(image, width=512)
274
275         # show the output image
276         cv2.imshow("Output", image)
277         cv2.waitKey(0)
```

We loop over the predicted probabilities + class labels on **Line 258** and proceed to draw both the label and corresponding score on the `image`.

Line 273 resizes our `image` such that it will fit on our screen while **Lines 276-277** show the final output.

The final code block, the "investigate" mode, is identical to Chapter 3 so I am going to omit the explanation from this chapter (the code block is still incorporated in the `pills.py` file included with your download of this book).

For a detailed explanation of the "investigate" code block used to debug the `load_pills`, `load_image`, and `load_mask` functions, please refer to Section 3.4.2.

4.2.3 Training the Mask R-CNN

Now that we've completed implementing the `pills.py` script, let's go ahead and train our Mask R-CNN. Open up a terminal and execute the following command:

```
$ python pills.py --mode train
...
Starting at epoch 0. LR=0.001
Epoch 1/10
30/30 - 39s 1s/step - loss: 1.4033 - val_loss: 0.8334
Epoch 2/10
30/30 - 36s 1s/step - loss: 0.6144 - val_loss: 0.4721
```

We start by training *just* the layer heads of the model. A single epoch is taking just ≈ 35 seconds on my Titan X GPU. Both training and validation loss steadily decrease as we train.

After the tenth epoch we unfreeze all layers and allow the entire network to train:

```
Starting at epoch 10. LR=0.0001
Epoch 11/20
30/30 - 50s 2s/step - loss: 0.1138 - val_loss: 0.1410
Epoch 12/20
30/30 - 45s 2s/step - loss: 0.0794 - val_loss: 0.1516
...
Epoch 19/20
30/30 - 45s 2s/step - loss: 0.0534 - val_loss: 0.1420
Epoch 20/20
30/30 - loss: 0.0480 - val_loss: 0.1436
```

Now that we are training the entire network each epoch is taking ≈ 45 seconds. Again, training loss and validation loss continue to drop, although training loss is now falling more sharply than before. I did not want to train past epoch 10 as I was concerned about overfitting with such a small dataset.

4.2.4 Segmenting Pills with Your Mask R-CNN

Now that our network is trained, let's make some predictions with it. Open up your terminal and execute the following command:

```
$ python pills.py --mode predict --image examples/pills_01.jpg
```

In Figure 4.3 you can see example outputs of our Mask R-CNN applied to pill segmentation. It's important to note that *none* of these images are part of the actual pills dataset the Mask R-CNN was trained on. Instead, I gathered these testing images from other websites on the internet.

As you can see, we have correctly segmented the pills from the images, which his especially impressive given the oblique viewing angle of some pills.

To further increase the accuracy of our pill segmentation system we would need additional (annotated) training data. Furthermore, we could also extend our pill segmentation system to non-round pills by providing such images during training.

I will leave that as an exercise to you, the reader, as a way to gain practice annotating data using an annotation tool from Section 4.1.3 above and then train your Mask R-CNN.

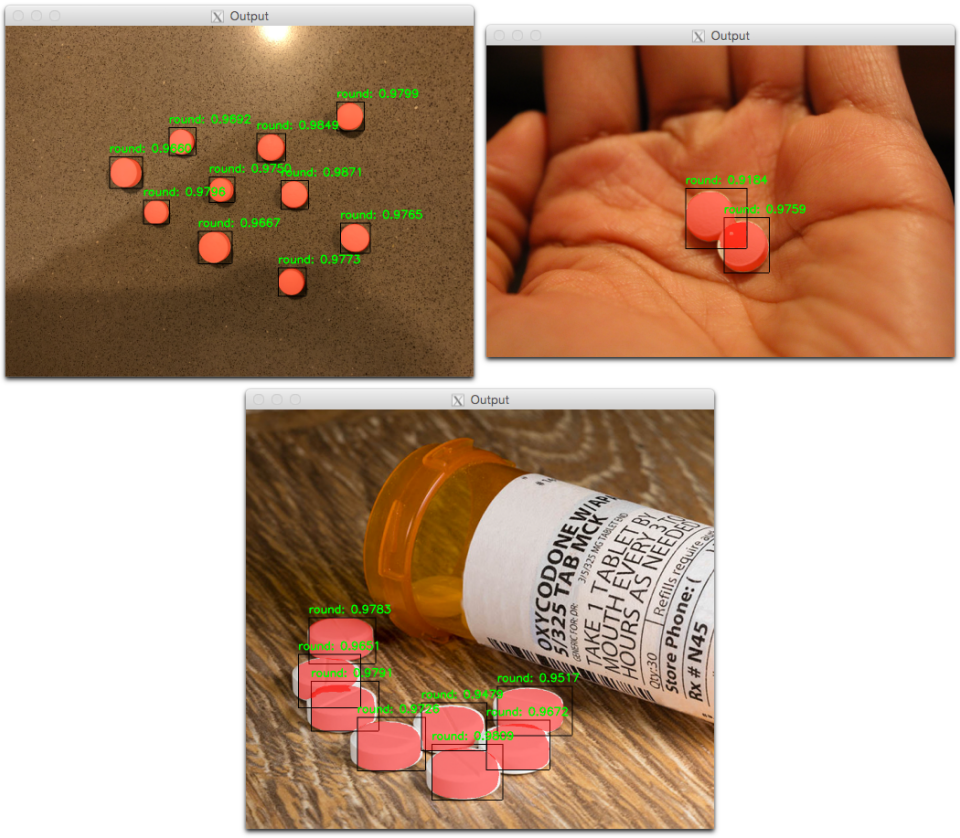


Figure 4.3: The results of applying our Mask R-CNN to a sample of images *outside* the dataset we trained our model here. Here we can see that our model is generalizing well. Further accuracy can be obtained by training the network with more images.

4.3 Summary

So far in our exploration of Mask R-CNNs we had only learned how to train a model on an *existing* dataset — but what if we wanted to train a Mask R-CNN *on our own dataset?*

In order to accomplish such a task, we first explored various image annotation tools that you may decide to leverage for instance segmentation. For the purposes of our project, we used my personal favorite, the VGG Image Annotator (VIA).

Once we had annotated all images in our pills dataset we then created a Python script that:

- Parsed our JSON file
- Constructed the masks for each image on the fly
- Successfully trained a Mask R-CNN for pill segmentation
- Was able to predict masks for pills *outside* of our original dataset

With additional training data our model can become more accurate as well.

Make sure you use the code in this chapter as a starting point for your own Mask R-CNN projects; however, keep in mind that *all* machine learning projects start with the quality of your data. While annotating your dataset is certainly the most tedious task, it's also the most important — don't skimp on your training data, it will only hurt you in the long run.

Bibliography

[1] Fizyr team and open source community. *keras-retinanet: Keras implementation of RetinaNet object detection.* `https://github.com/fizyr/keras-retinanet`. 2017 (cited on pages 9, 10, 20, 21, 23, 30).

[2] TensorFlow Community. *Tensorflow Object Detection API*. `https://github.com/tensorflow/models/tree/master/research/object_detection`. 2017 (cited on page 9).

[3] Tsung-Yi Lin et al. "Focal Loss for Dense Object Detection". In: *CoRR* abs/1708.02002 (2017). URL: `http://arxiv.org/abs/1708.02002` (cited on pages 9, 10, 21, 23, 30).

[4] Frank Fotso. *Application of py_faster_rcnn in logo detection task: ZF & VGG16*. `https://github.com/franckfotso/faster_rcnn_logo/blob/master/README.md`. 2017 (cited on page 10).

[5] Tsung-Yi Lin et al. "Microsoft COCO: Common Objects in Context". In: *CoRR* abs/1405.0312 (2014). URL: `http://arxiv.org/abs/1405.0312` (cited on pages 12, 41, 66).

[6] M. Everingham et al. "The Pascal Visual Object Classes (VOC) Challenge". In: *International Journal of Computer Vision* 88.2 (June 2010), pages 303–338 (cited on page 14).

[7] Joseph Redmon et al. "You Only Look Once: Unified, Real-Time Object Detection". In: *CoRR* abs/1506.02640 (2015). URL: `http://arxiv.org/abs/1506.02640` (cited on page 23).

[8] Wei Liu et al. "SSD: Single Shot MultiBox Detector". In: *CoRR* abs/1512.02325 (2015). URL: `http://arxiv.org/abs/1512.02325` (cited on page 23).

[9] Shaoqing Ren et al. "Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks". In: *CoRR* abs/1506.01497 (2015). URL: `http://arxiv.org/abs/1506.01497` (cited on pages 23, 35, 36, 57).

[10] Roberto Olmos, Siham Tabik, and Francisco Herrera. "Automatic Handgun Detection Alarm in Videos Using Deep Learning". In: *CoRR* abs/1702.05147 (2017). URL: `http://arxiv.org/abs/1702.05147` (cited on pages 23, 24).

[11] Leonard Richardson. *Beautiful Soup: We called him Tortoise because he taught us.* https://www.crummy.com/software/BeautifulSoup/. 2004 (cited on page 27).

[12] Kaiming He et al. "Mask R-CNN". In: *CoRR* abs/1703.06870 (2017). URL: http://arxiv.org/abs/1703.06870 (cited on pages 33, 35, 36, 57).

[13] Matterport Inc. and community. *Matterport - Keras + Maks R-CNN*. 2017 (cited on pages 33, 38, 57).

[14] Alberto Garcia-Garcia et al. "A Review on Deep Learning Techniques Applied to Semantic Segmentation". In: *CoRR* abs/1704.06857 (2017). URL: http://arxiv.org/abs/1704.06857 (cited on page 34).

[15] Jianbo Shi and Jitendra Malik. "Normalized Cuts and Image Segmentation". In: *IEEE Trans. Pattern Anal. Mach. Intell.* 22.8 (Aug. 2000), pages 888–905. ISSN: 0162-8828 (cited on page 34).

[16] Yuri Boykov and Gareth Funka-Lea. "Graph Cuts and Efficient N-D Image Segmentation". In: *Int. J. Comput. Vision* 70.2 (Nov. 2006), pages 109–131. ISSN: 0920-5691 (cited on page 34).

[17] Carsten Rother, Vladimir Kolmogorov, and Andrew Blake. ""GrabCut": Interactive Foreground Extraction Using Iterated Graph Cuts". In: *ACM Trans. Graph.* 23.3 (Aug. 2004), pages 309–314. ISSN: 0730-0301 (cited on page 34).

[18] Pedro F. Felzenszwalb and Daniel P. Huttenlocher. "Efficient Graph-Based Image Segmentation". In: *Int. J. Comput. Vision* 59.2 (Sept. 2004), pages 167–181. ISSN: 0920-5691 (cited on page 34).

[19] Andrea Vedaldi and Stefano Soatto. "Quick shift and kernel methods for mode seeking". In: *In European Conference on Computer Vision, volume IV*. 2008, pages 705–718 (cited on page 34).

[20] Radhakrishna Achanta et al. "SLIC Superpixels Compared to State-of-the-Art Superpixel Methods". In: *IEEE Trans. Pattern Anal. Mach. Intell.* 34.11 (Nov. 2012), pages 2274–2282. ISSN: 0162-8828 (cited on page 34).

[21] P. Neubert and P. Protzel. "Compact Watershed and Preemptive SLIC: On Improving Trade-offs of Superpixel Segmentation Algorithms". In: *2014 22nd International Conference on Pattern Recognition*. Aug. 2014, pages 996–1001 (cited on page 34).

[22] Ross B. Girshick et al. "Rich feature hierarchies for accurate object detection and semantic segmentation". In: *CoRR* abs/1311.2524 (2013). URL: http://arxiv.org/abs/1311.2524 (cited on page 35).

[23] Ross B. Girshick. "Fast R-CNN". In: *CoRR* abs/1504.08083 (2015). arXiv: 1504.08083. URL: http://arxiv.org/abs/1504.08083 (cited on page 35).

[24] Adrian Rosebrock. *(Faster) Non-Maximum Suppression in Python*. 2015 (cited on page 36).

[25] Noel C. F. Codella et al. "Skin Lesion Analysis Toward Melanoma Detection: A Challenge at the 2017 International Symposium on Biomedical Imaging (ISBI), Hosted by the International Skin Imaging Collaboration (ISIC)". In: *CoRR* abs/1710.05006 (2017). URL: http://arxiv.org/abs/1710.05006 (cited on pages 38, 39, 57, 59).

[26] Philipp Tschandl, Cliff Rosendahl, and Harald Kittler. "The HAM10000 Dataset: A Large Collection of Multi-Source Dermatoscopic Images of Common Pigmented Skin Lesions". In: *CoRR* abs/1803.10417 (2018). URL: http://arxiv.org/abs/1803.10417 (cited on pages 38, 39, 57, 59).

[27] imgaug Open Source Community. *imgaug - Image augmenation for machine learning experiments*. 2015 (cited on page 42).

[28] Z. Yaniv et al. “The national library of medicine pill image recognition challenge: An initial report”. In: *2016 IEEE Applied Imagery Pattern Recognition Workshop (AIPR)*. Oct. 2016, pages 1–9 (cited on pages 59, 60).

[29] Davis E. King. “Dlib-ml: A Machine Learning Toolkit”. In: *J. Mach. Learn. Res.* 10 (Dec. 2009), pages 1755–1758. ISSN: 1532-4435. URL: http://dl.acm.org/citation.cfm?id=1577069.1755843 (cited on page 61).

[30] A. Dutta, A. Gupta, and A. Zissermann. *VGG Image Annotator (VIA)*. http://www.robots.ox.ac.uk/~vgg/software/via/. 2016 (cited on page 62).