3

Data Preparation for Object Detection Applications

This chapter discusses the steps to prepare data for training models using Detectron2. Specifically, it provides tools to label images if you have some datasets at hand. Otherwise, it points you to places with open datasets so that you can quickly download and build custom applications for computer vision tasks. Additionally, this chapter covers the techniques to convert standard annotation formats to the data format required by Detectron2 if the existing datasets come in different formats.

By the end of this chapter, you will know how to label data for object detection tasks and how to download existing datasets and convert data of different formats to the format supported by Detectron2. Specifically, this chapter covers the following:

- Common data sources
- Getting images
- Selecting image labeling tools
- Annotation formats
- Labeling the images

- Annotation format conversions

Technical requirements

You should have set up the development environment with the instructions provided in ***Chapter 1***. Thus, if you have not done so, please complete setting up the development environment before continuing. All the code, datasets, and results are available on the GitHub page of the book (under the folder named `Chapter03`) at **https://github.com/PacktPublishing/Hands-On-Computer-Vision-with-Detectron2**. It is highly recommended to download the code and follow along.

Common data sources

Chapter 2 introduced the two most common datasets for the computer vision community. They are **ImageNet** and Microsoft **COCO** (**Common Objects in Context**). These datasets also contain many pre-trained models that can predict various class labels that may meet your everyday needs.

If your task is to detect a less common class label, it might be worth exploring the **Large Vocabulary Instance Segmentation** (**LVIS**) dataset. It has more than 1,200 categories and 164,000 images, and it contains many rare categories and about 2 million high-quality instance segmentation masks. Detectron2 also provides pre-trained models for predicting these

1,200+ labels. Thus, you can follow the steps described in ***Chapter 2*** to create a computer vision application that meets your needs. More information about the LVIS dataset is available on its website at **https://www.lvisdataset.org**.

If you have a task where no existing/pre-trained models can meet your needs, it is time to find existing datasets and train your model. One of the common places to look for datasets is **Kaggle** (**https://www.kaggle.com**). Kaggle is a Google subsidiary. It provides an online platform for data scientists and machine learning practitioners to share datasets, train machine learning models, and share the trained models and their thoughts via discussions. It also often holds and allows users to host data science and machine learning competitions (including computer vision competitions, of course). At the time of writing, searching the phrase "object detection" on Kaggle brings 555 datasets, as shown in *Figure 3.1*.

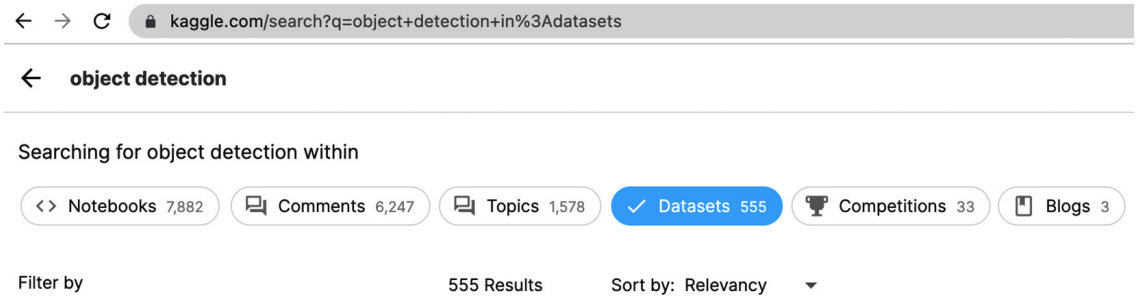

Figure 3.1: Datasets on Kaggle for object detection tasks

Kaggle contains datasets for data science and machine learning in general, including computer vision datasets. However, if you still cannot find a dataset for your custom needs, you may be interested in exploring

Roboflow Universe available at **https://universe.roboflow.com**. It claims to be the world's most extensive collection of open source computer vision datasets and APIs. At the time of writing, more than 100 million images, 110,000 datasets, and 10,000 pre-trained models are available on Roboflow Universe, as shown in *Figure 3.2*. Datasets for common computer vision tasks are available, such as object detection, instance segmentation, and semantic segmentation.

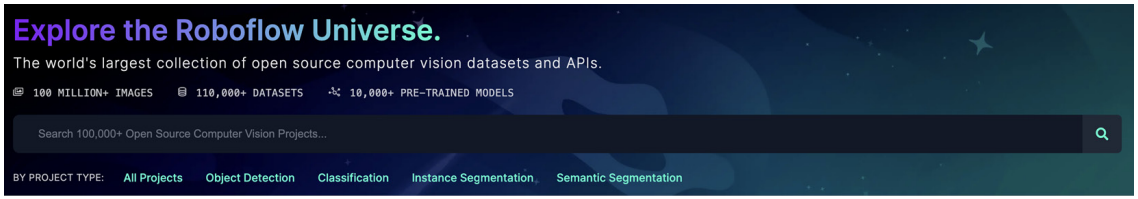

Figure 3.2: Datasets on Roboflow Universe

Computer vision is advancing rapidly, and there is a vast number of available practitioners in this field. If you cannot find the datasets that meet your needs in the preceding two resources, you can use the internet (e.g., Google) to search for the dataset that you want. However, if you still cannot find such a dataset, it is time to collect and label your own dataset. The following sections cover the standard tools and steps for collecting and labeling datasets for computer vision applications.

Getting images

When you start a new project, you might have a set of images collected using graphic devices such as cameras, smartphones, drones, or other

specialized devices. If you have some photos, you can skip this step and move to the following steps, where you select a labeling tool and start labeling. However, if you are about to start a fresh new project and do not even have any images, you can get some ideas from this section on collecting pictures from the internet. There might be copyrights related to images that you collect from the internet. Thus, you should check for copyrights to see whether the downloaded images are usable for your purposes.

Google Images contains a massive number of photos, which continues to grow every day. Therefore, it is a great resource to crawl images for training your custom model. Python has a **`simple_image_download`** package that provides scripts to search for images using keywords or tags and download them. First, we will install this package using the following command:

```
!pip install simple_image_download
```

You can run this command in Google Colab's code cell to install it and download images to a running Google Colab instance. However, once the instance is stopped, the downloaded photos are gone. Therefore, you should download the images to a mapped Google Drive folder or on a local computer. You can run the same command on a locally hosted Jupyter notebook or remove the start (`!`) character and execute it in a terminal to install this package locally. If you are running on Google Colab, the following code snippet helps to map Google Drive and creates a folder in Google Drive:

```
import os
from google.colab import drive
drive.mount('/gdrive')
project_folder = "/gdrive/My Drive/Detectron2/Chapter3"
os.makedirs(project_folder, exist_ok=True)
```

The next code snippet (to be executed in a separate Google Colab code cell) sets the working directory to the previously created folder, so the downloaded results remain in this folder when the Google Colab instance stops:

```
cd "/gdrive/My Drive/Detectron2/Chapter3"
```

It is always a good practice to ensure that Google Colab is currently working in this directory by running the **`pwd()`** command in another code cell and inspecting the output.

Once you are in the specified folder, the following code snippet helps to download images from Google Images using the **`simple_image_download`** package:

```
from tqdm import tqdm
from simple_image_download.simple_image_download import simple_image_download as Downloader
def _download(keyword, limit):
  downloader = Downloader()
  downloader.download(keywords=keyword, limit=limit)
  urls = downloader.urls(keywords=keyword, limit=limit)
  return urls
```

```
def download(keywords, limit):
  for keyword in tqdm(keywords):
    urls = _download(keyword=keyword, limit=limit)
    with open(f"simple_images/{keyword}.txt", "w") as f:
      f.writelines('\n'.join(urls))
```

Specifically, the first **`import`** statement helps to import the **`tqdm`** package, which can be used to show downloads' progress. This is important because downloading a large number of images from the internet would take time. Therefore, it would be helpful to have a progress bar showing us that it is currently working and approximately how long it will take to complete the task. The next **`import`** statement imports the **`simple_image_download`** object, which helps to perform the downloading task. The next is the **`_download`** helper method to download a number (specified by the **`limit`** parameter) of images from Google using a keyword (specified by the **`keyword`** parameter). Furthermore, for the corresponding pictures, this helper method also returns an array of URLs that can be stored to validate the downloaded images and check for copyrights if needed.

The next method is called **`download`**. It uses the previous helper to download images, and the results are stored under a folder called **`simple_images`**. The images for each keyword are stored in a subfolder named after the keyword itself. Furthermore, this method writes a text file per keyword for all the corresponding URLs to the downloaded images with the same name as the keyword plus the **`.txt`** extension. For instance, the following code snippet downloads 10 images for the **`brain tumors x-ray`** and **`heart tumors x-ray`** keywords:

```
download(keywords=["brain tumors x-ray", "heart tumors x-ray"], limit=10)
```

Figure 3.3 shows the resulting folders in Google Drive for the downloaded images and text files to store the image URLs on the internet. These URLs can be used to validate the images or check for copyrights.

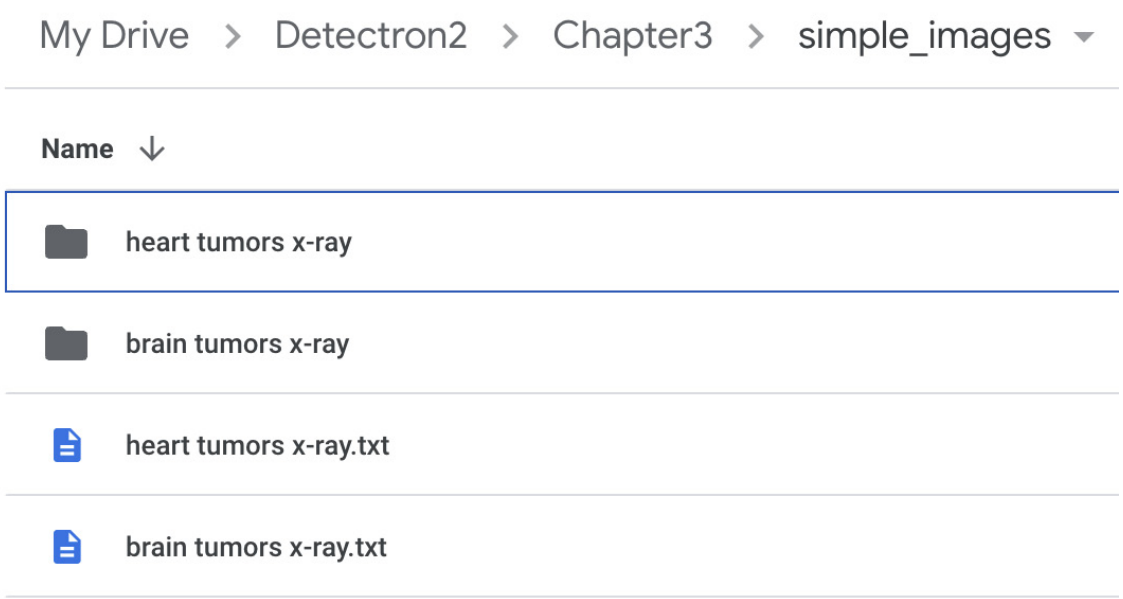

Figure 3.3: Resulting image folders and text files for their corresponding URLs

The downloaded images may not be what you are looking for exactly. Therefore, you should check the downloaded images and delete those that do not meet your specific needs. Once you have the desired images, the next step is to select a tool to start labeling the images for training and validation.

Selecting an image labeling tool

Computer vision applications are developing rapidly. Therefore, there are many tools for labeling images for computer vision applications. These tools range from free and open source to fully commercial or commercial with free trials (which means there are some limitations regarding available features). The labeling tools may be desktop applications (require installations), online web applications, or locally hosted web applications. The online applications may even provide cloud storage, utilities to support collaborative team labeling, and pre-trained models that help to label quicker (generally, with some cost).

One of the popular image labeling tools is `labelImg`. It is available at **https://github.com/heartexlabs/labelImg/blob/master/README.rst**. It is an open source, lightweight, fast, easy-to-use, Python-based application with a short learning curve.

Its limitation is that it supports only the creation of rectangle bounding boxes. It is currently part of **Label Studio** (which is a more complicated tool with many more features). However, it is still the best choice if the task at hand is to perform object detection and localization only.

Another popular tool is `labelme` (**https://github.com/wkentaro/labelme**). It is similar to `labelImg` in terms of being a Python-based application, open source, fast, and easy to use. However, it supports other annotation types besides rectangle bounding boxes. Specifically, it can annotate poly-

gons, rectangles, circles, lines, and points. Therefore, it can be used to label training images for object detection and semantic or instance segmentation tasks.

Both `labelImg` and `labelme` are locally installed applications. However, if you want to work with a locally hosted web application, you may refer to **VGG (Visual Geometry Group) Image Annotator (VIA)**. VIA is a lightweight, standalone, browser-based web application. It also supports annotating polygons, circles, lines, and rectangular bounding boxes.

If you want to use a cloud-based web application (that supports cloud storage and annotation on the web), you may like `roboflow`, available at **https://app.roboflow.com**. This online web application supports various annotation modes. It has utilities for teams to label images collaboratively. It also supports pre-trained models to help label images faster. However, several of its advanced features are only available in the paid plan. Another popular web platform for this task is **https://www.make-sense.ai**. It is open source and free to use under a GPLv3 license. It supports several popular annotation formats and uses AI to make your labeling job more productive.

This chapter selects `labelImg` to prepare data for the object detection tasks. ***Chapter 10*** uses `labelme` to prepare data for the object instance segmentation tasks. After choosing a labeling tool, the next step is to discover which annotation formats that tool supports, which one to select, and how to convert it to the format that Detectron2 supports.

Annotation formats

Similar to labeling tools, many different annotation formats are available for annotating images for computer vision applications. The common standards include COCO JSON, Pascal VOC XML, and YOLO PyTorch TXT. There are many more formats (e.g., TensorFlow TFRecord, CreateML JSON, and so on). However, this section covers only the previously listed three most common annotation standards due to space limitations. Furthermore, this section uses two images and labels extracted from the test set of the brain tumor object detection dataset available from Kaggle (**https://www.kaggle.com/datasets/davidbroberts/brain-tumor-object-detection-datasets**) to illustrate these data formats and demonstrate their differences, as shown in *Figure 3.4*. This section briefly discusses the key points of each annotation format, and interested readers can refer to the GitHub page of this chapter to inspect this same dataset in different formats in further detail.

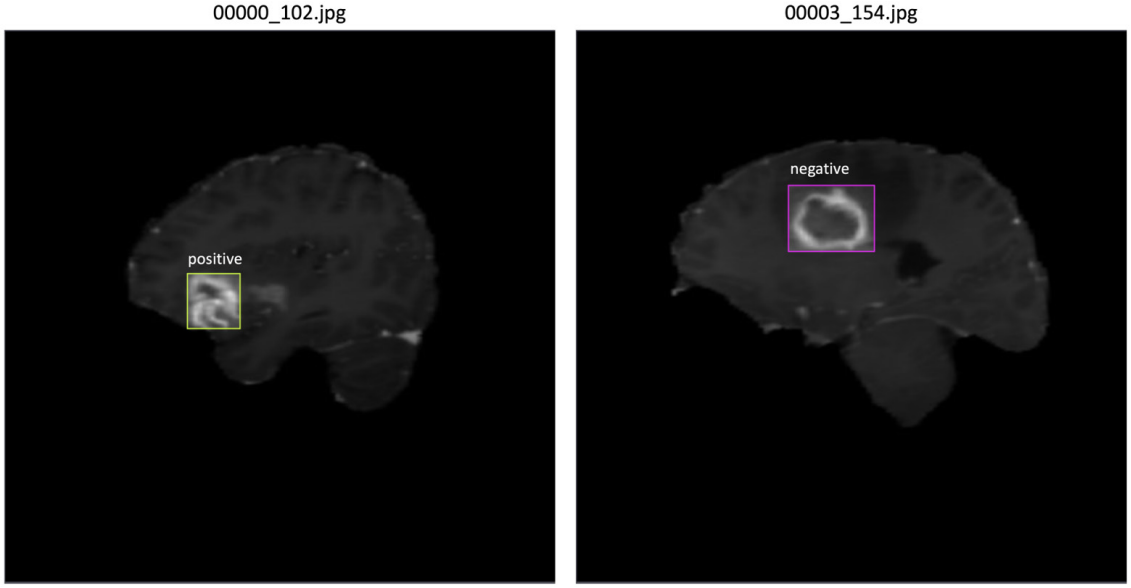

Figure 3.4: Two images and tumor labels used to illustrate different annotation formats

The COCO **JSON** (**JavaScript Object Notation**) format is famous thanks to the popularity of the Microsoft COCO dataset. The Detectron2 data format is based on COCO's annotation standard. Therefore, it is worth diving into this dataset a little bit deeper. Each dataset in COCO's annotation format is represented in one JSON file. This JSON file is stored in the same folder with all the images. Thus, it is often named with "_" as the start character to bring it to the top of the list view when viewing this data folder. COCO's annotation JSON file contains these main elements:

- The info section (`info`) provides general descriptions of the dataset
- The licenses section (`licenses`) is a list of licenses applied to the images
- The categories section (`categories`) is a list of the available categories (or class labels) and supercategories for this dataset

- The images section (**`images`**) is a list of image elements, and each has information such as the **`id`**, **`width`**, **`height`**, and **`file_name`** of the image
- The annotations section (**`annotations`**) is a list of annotations, and each annotation has information such as segmentation and a bounding box (**`bbox`**)

The following snippet provides sample sections of a dataset with two images in the test set (available from **https://www.kaggle.com/datasets/davidbroberts/brain-tumor-object-detection-datasets**) in COCO annotation format:

Here is the **`info`** section:

```
"info": {
        "year": "2022",
        "version": "1",
        "description": "Brain tumor object detection",
        "contributor": "",
        "url": "<the URL to Kaggle dataset>",
        "date_created": "2022-10-07T13:54:09+00:00"
    },
```

This is the **`licenses`** section:

```
"licenses": [
        {
            "id": 1,
            "url": "https://creativecommons.org/publicdomain/zero/1.0/",
```

```
            "name": "CC0 1.0"
        }
    ],
```

This is the **`categories`** section:

```
  "categories": [
        {
            "id": 0,
            "name": "tumors",
            "supercategory": "none"
        },
        {
            "id": 1,
            "name": "negative",
            "supercategory": "tumors"
        },
        {
            "id": 2,
            "name": "positive",
            "supercategory": "tumors"
        }
    ],
```

Here is the **`images`** section:

```
    "images": [
        {
            "id": 0,
            "license": 1,
```

```
            "file_name": "00000_102.jpg",
            "height": 512,
            "width": 512,
            "date_captured": "2022-10-07T13:54:09+00:00"
        },
        {
            "id": 1,
            "license": 1,
            "file_name": "00003_154.jpg",
            "height": 512,
            "width": 512,
            "date_captured": "2022-10-07T13:54:09+00:00"
        }
    ],
```

Here is the **`annotations`** section:

```
    "annotations": [
        {
            "id": 0,
            "image_id": 0,
            "category_id": 2,
            "bbox": [170,226,49,51],
            "area": 2499,
            "segmentation": [],
            "iscrowd": 0
        },
        {
            "id": 1,
            "image_id": 1,
```

```
            "category_id": 1,
            "bbox": [197,143,80,62],
            "area": 4960,
            "segmentation": [],
            "iscrowd": 0
        }
    ]
```

Notably, the bounding box is specified in absolute pixel locations. The co-ordinate system has the height as the *y* axis, the width as the *x* axis, and the origin at the top-left corner of the picture. For instance, for the first image (**`00003_154.jpg`** and **`image_id = 1`**), the bounding box is **`[197,143,80,62]`**, which are the values for **`[x_min, y_min, width, height]`**, respectively, as shown in *Figure 3.5*.

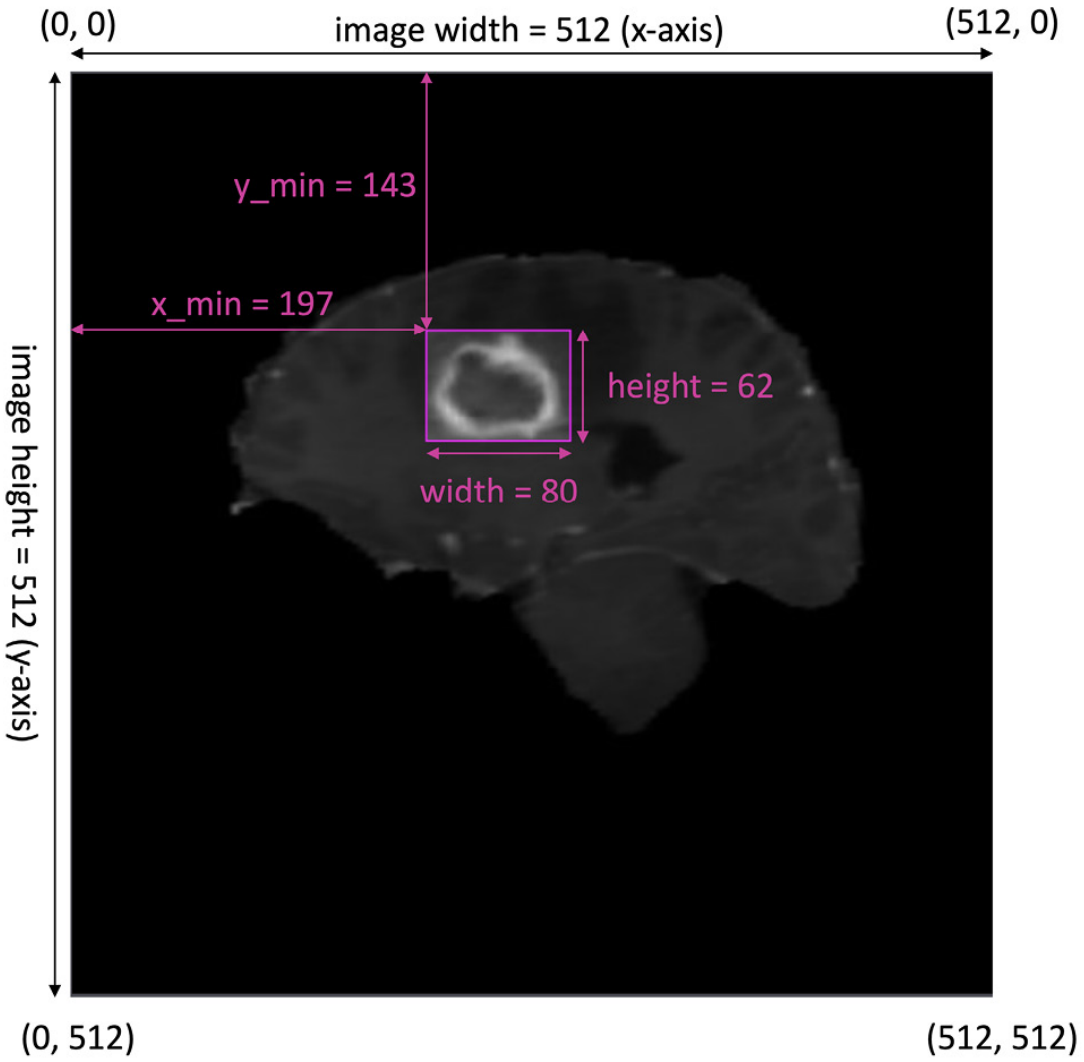

Figure 3.5: Coordinate system and annotation format ([x_min, y_min, width, height]) in COCO style for bounding box

Another popular image annotation format is the **Pascal VOC** (**Pattern Analysis, Statistical Modeling, and Computational Learning Visual Object Classes**) format. This annotation format is in the **XML** (**eXtensible Markup Language**) file format, and there is one XML file per image.

The annotation filename has the same name as the image filename with the `.xml` extension instead. Storing one XML file per image prevents having one large file for all the annotations for a large image dataset. The following is the annotation file (`00003_154.xml`) for one image in the demo dataset (`00003_154.jpg`):

```
<annotation>
    <folder></folder>
    <filename>00003_154.jpg</filename>
    <path>00003_154.jpg</path>
    <source>
        <database>annotationdemo</database>
    </source>
    <size>
        <width>512</width>
        <height>512</height>
        <depth>3</depth>
    </size>
    <segmented>0</segmented>
    <object>
        <name>negative</name>
        <pose>Unspecified</pose>
        <truncated>0</truncated>
        <difficult>0</difficult>
        <occluded>0</occluded>
        <bndbox>
            <xmin>197</xmin>
            <xmax>277</xmax>
            <ymin>143</ymin>
```

```
            <ymax>205</ymax>
        </bndbox>
    </object>
</annotation>
```

Observably, this annotation file contains image information and annotation objects. As shown in *Figure 3.6*, the image coordinate system is the same as described previously for the COCO annotation format. It also has four numbers for the bounding box: **`xmin`**, **`xmax`**, **`ymin`**, and **`ymax`**. In other words, the bounding box is stored using the top-left and bottom-right coordinates in absolute pixel values (instead of the top-left corner and width and height values).

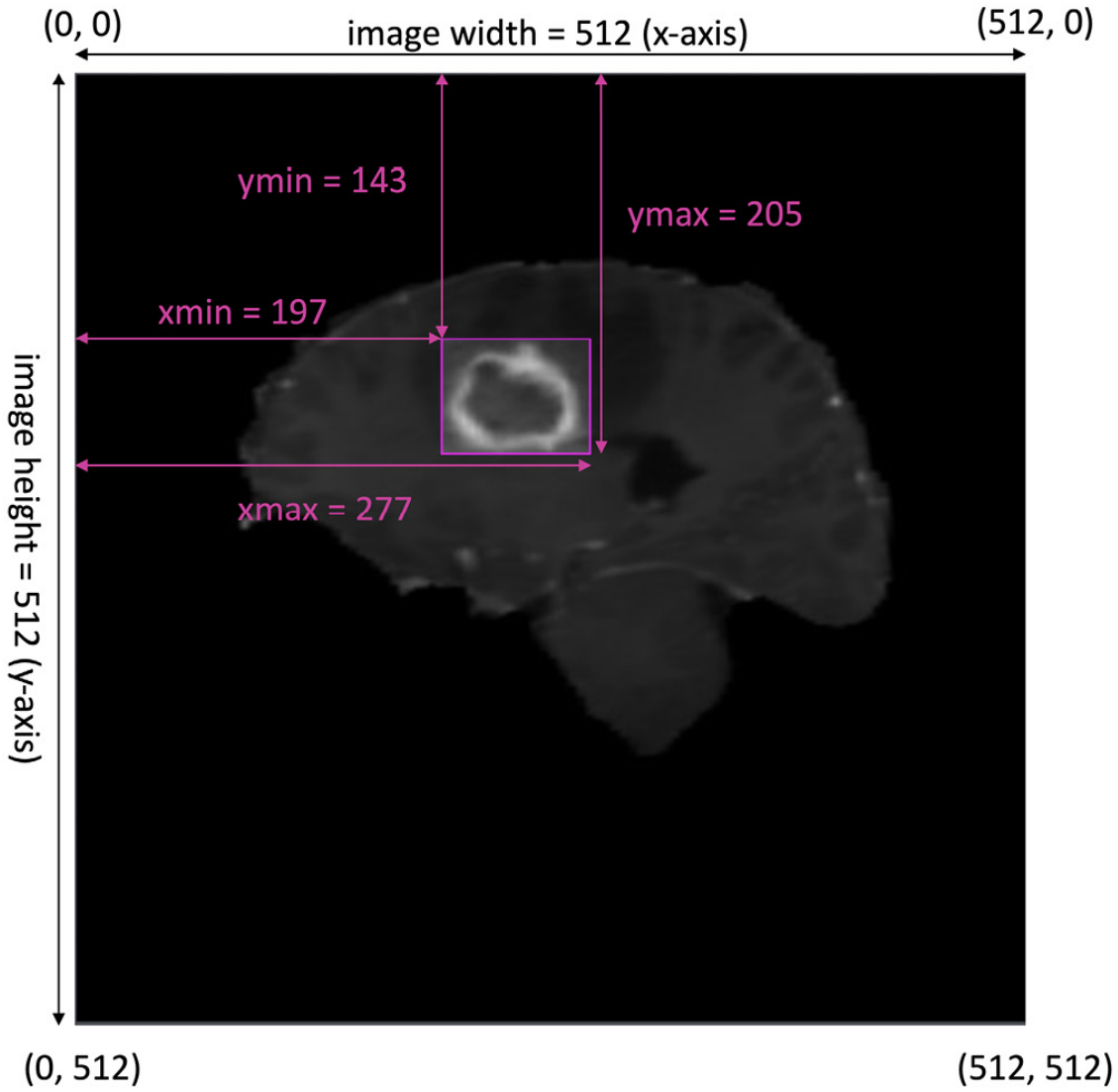

Figure 3.6: Coordinate system and annotation format ([xmin, xmax, ymin, ymax]) in Pascal VOC style for bounding box

Due to the recent development of **YOLO** (**You Only Look Once**), YOLO data annotation is another popular data format. Similar to the PASCAL

VOC annotation style, the dataset has one annotation file per image. However, the annotation file extension is **`.txt`** instead.

A dataset is organized into two subfolders, one for images (**`images`**) and one for annotations (**`labels`**). These subfolders might be further divided into the train (**`train`**), test (**`test`**), and validation (**`val`**) subfolders. There is also a text file called **`classes.txt`** (or possibly a **`*.yaml`** file) that declares the classes or labels for this dataset, one per line. The following is the content of the **`classes.txt`** file for our simple demo dataset with two classes:

```
negative
positive
```

Different from the COCO and Pascal VOC annotation styles, YOLO stores one annotation per line. Specifically, there are five values in a line per bounding box. They are the label index (**`label_idx`**), the center coordinates (**`center_x, center_y`**), and then the width (**`width`**) and height (**`height`**) of the bounding box. The label index is the map of the label into its corresponding index (starting from **`0`** for the first label). For instance, in this specific case, **`negative`** is mapped into **`0`**, and **`positive`** is mapped into **`1`**.

For example, the annotation for the **`00003_154.jpg`** image is **`00003_154.txt`**, and it has the following content:

```
0 0.463028 0.339789 0.156103 0.120892
```

These values correspond to **`[label_idx, center_x, center_y, width, height]`**, as described, for one bounding box. Notably, the coordinate of the center, width, and height of the bounding box are all converted into values as ratios of the corresponding values to the image size. *Figure 3.7* illustrates this annotation style for the **`00003_154.jpg`** image.

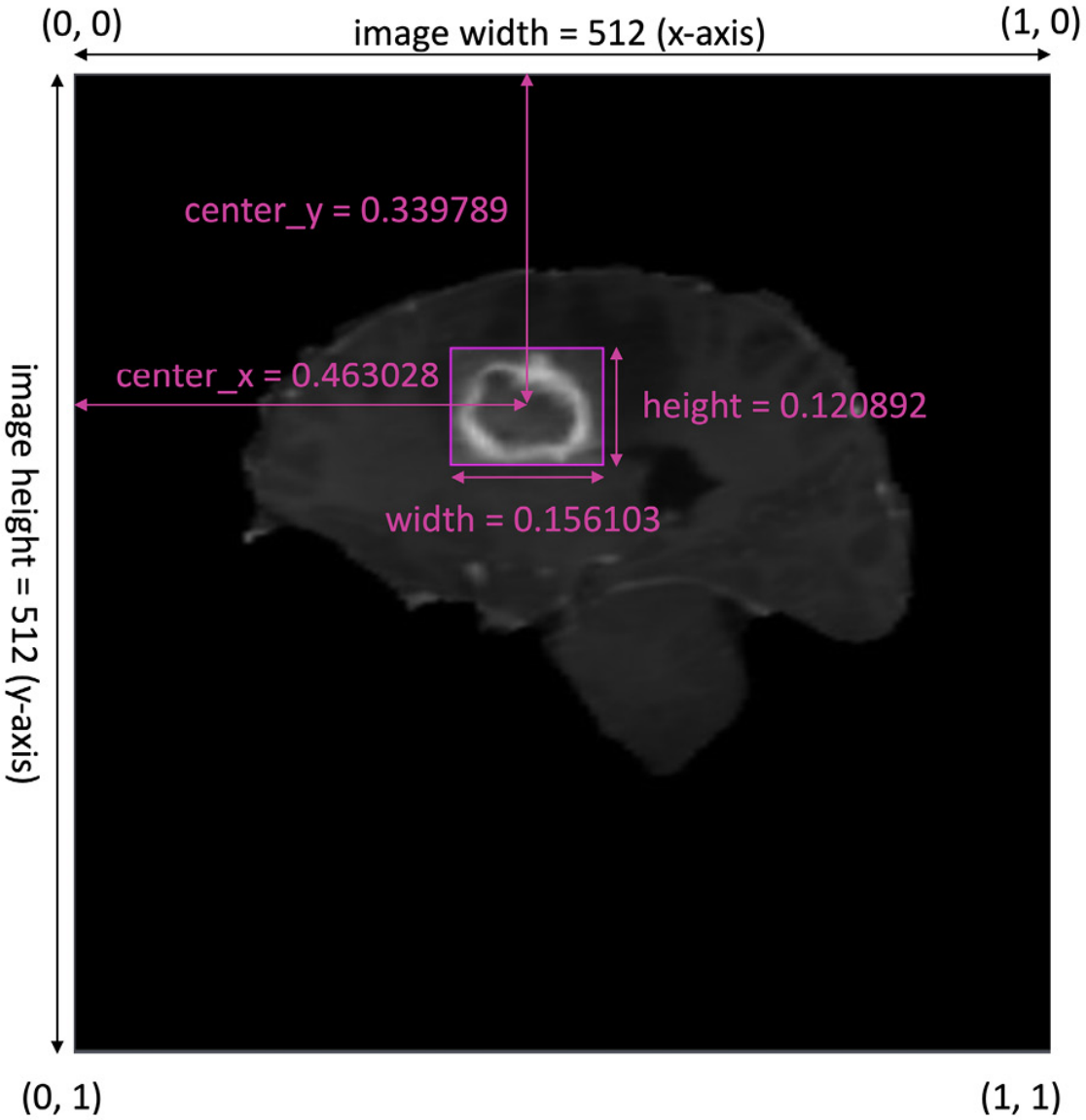

Figure 3.7: Coordinate system and annotation format ([center_x, center_y, width, height]) in YOLO style for bounding box

Congratulations! At this point, you should clearly understand the essential and popular annotation formats used by the computer vision commu-

nity to annotate images. The next step is to label a dataset for your custom computer vision task. The following section guides you through the steps that use **`labelImg`** to prepare data for the object detection task. ***Chapter 10*** uses **`labelme`** to prepare data for the object instance segmentation tasks.

Labeling the images

This chapter uses **`labelImg`** to perform data labeling for object detection tasks. This tool requires installation on the local computer. Therefore, if you downloaded your images using Google Colab to your Google Drive, you need to map or download these images to a local computer to perform labeling. Run the following snippet in a terminal on a local computer to install **`labelImg`** using Python 3 (if you are running Python 2, please refer to the **`labelImg`** website for a guide):

```
pip3 install labelImg
```

After installing, run the following snippet to start the tool, where **`[IMAGE_PATH]`** is an optional argument to specify the path to the image folder, and **`[PRE-DEFINED CLASS FILE]`** is another optional argument to indicate the path to a text file (**`*.txt`**) that defines a list of class labels (one per line):

```
labelImg [IMAGE_PATH] [PRE-DEFINED CLASS FILE]
```

For instance, after downloading/synchronizing the **`simple_images`** folder for the downloaded images in the previous steps, we can set the working directory of the terminal to this directory and use the following command to start labeling the pictures on the "**`brain tumors x-ray`**" folder:

```
labelImg "brain tumors x-ray"
```

Figure 3.8 shows the main interface of **`labelImg`** with the main menu on the left, the list of pictures on the bottom right, the current labeling image at the center, and the list of labels on this current image in the top-right corner.

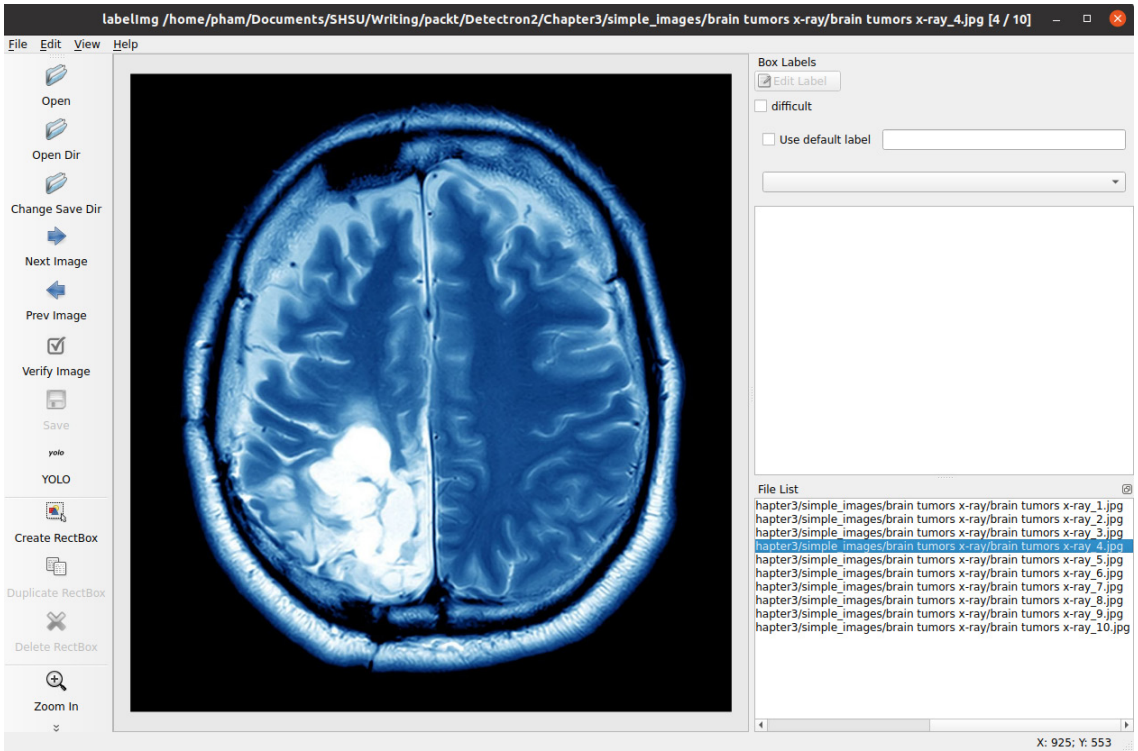

Figure 3.8: The labelImg interface for bounding box labeling

This tool supports exporting the data annotations into standard data formats such as Pascal VOC and YOLO. After installing and launching the application (using the preceding instructions), the following are the steps to label images using the Pascal VOC format:

1. Click on the **Open Dir** menu to open the image directory (if you did not specify the image directory at launching time).
2. Click on the **Change Save Dir** menu to change the annotation directory.
3. Click on **Create RectBox** to start annotating rectangular boxes.
4. Click and release the left mouse button to select a region to annotate the bounding box. You can use the right mouse button to drag the rectangle box to copy or move it.

IMPORTANT NOTE

Please refer to the previous section regarding the folder structure of YOLO to create corresponding folders for images and labels before starting labeling using the following steps.

Similarly, the following are the steps to label images using the YOLO format:

1. Create a file called **`classes.txt`** with a list of labels (one label per line).
2. Launch **`labelImg`** using the previous instructions.
3. Click on the **Open Dir** menu to open the image directory (if you did not specify the image directory at launching time).

4. If the current format is not YOLO (it might currently be CreateML or Pascal VOC), click on the **CreateML** or **Pascal VOC** menu until it switches to **YOLO**.
5. Click on the **Change Save Dir** menu to change the annotation directory.

A **`.txt`** file per image with annotation in the YOLO format is saved in the selected directory with the same name. A file named **`classes.txt`** is saved in that folder too.

Additionally, the following shortcuts help support quick labeling:

- *The W* key: Create a bounding box
- *The D* key: Move to the next image
- *The A* key: Move to the previous image
- *Delete* key: delete the selected bounding box
- Arrow keys: Move the selected bounding box
- *Ctrl* + *Shift* + *D*: Delete the current image

Acquiring datasets from different sources or data labeling tools may come with annotation formats that differ from what Detectron2 supports (the COCO data format). Therefore, knowing how to convert datasets from different formats to the COCO format is essential. The following section guides you through performing this dataset conversion process.

Annotation format conversions

Detectron2 has its data description built based on the COCO annotation format. In other words, it supports registering datasets using the COCO data annotation format. However, other data annotation formats are abundant, and you may download a dataset or use a labeling tool that supports another data format different from COCO. Therefore, this section covers the code snippets used to convert data from the popular Pascal VOC and YOLO formats to COCO style.

IMPORTANT NOTE

A statement that starts with an exclamation mark (`!`) means it is a Bash command to be executed in a Jupyter notebook (Google Colab) code cell. If you want to run it in a terminal, you can safely remove this exclamation mark and execute this statement.

By understanding the different data formats as described, it is relatively easy to write code to convert data from one format to another. However, this section uses the **`pylabel`** package to perform this conversion to speed up the development time. In a Jupyter notebook or Google Colab code cell, run the following command to install the **`pylabel`** package:

```
!pip install pylabel
```

Converting YOLO datasets to COCO datasets

This section also uses the simple annotation demo dataset described previously to demonstrate how these conversions work. Therefore, you can

upload the YOLO and Pascal VOC datasets to Google Colab (if you are using Google Colab) or set the working directory of your local Jupyter notebook instance to the dataset folder (**annotationdemo**). The following is the code snippet for downloading the **yolo.zip** file, which stores the dataset in YOLO format into the current Google Colab running instance and unzips it:

```
!wget url_to_yolo.zip
!unzip yolo.zip
```

Optionally, you can install the **tree** command and list this simple dataset to view its structure using this snippet:

```
!sudo apt-get install tree
!tree yolo
```

The following is the sample output of this statement for this dataset:

```
yolo
├ classes.txt
├ images
│   └ test
│       ├ 00000_102.jpg
│       └ 00003_154.jpg
└ labels
    └ test
        ├ 00000_102.txt
        └ 00003_154.txt
```

The following snippet helps create a new COCO format dataset structure:

```
import os
import shutil
from glob import glob
from tqdm import tqdm
annotations_path = "yolo/labels/test"
images_path = "yolo/images/test"
coco_dir = 'coco/test'
os.makedirs(coco_dir, exist_ok=True)
txt_files = glob(os.path.join(annotations_path, "*.txt"))
img_files = glob(os.path.join(images_path, "*.jpg"))
# copy annotations
for f in tqdm(txt_files):
  shutil.copy(f, coco_dir)
# copy images
for f in tqdm(img_files):
  shutil.copy(f, coco_dir)
```

Specifically, COCO datasets typically have images placed in a folder. Therefore, we create a folder and place images into this folder. Additionally, this snippet copies the YOLO annotation files to this folder to perform the conversion. Once the conversion is completed, these files are removed.

The following snippet reads the classes from the YOLO dataset (**`classes.txt`**) and imports the YOLO annotations copied to the COCO dataset directory previously (**`coco_dir`**) to a **`pylabel`** dataset:

```
from pylabel import importer
# get the classes
with open("yolo/classes.txt", "r") as f:
  classes = f.read().split("\n")
# load dataset
dataset = importer.ImportYoloV5(path=coco_dir, cat_names=classes, name="brain tumors")
```

Notably, at the time of writing, YOLO v7 is available; however, it has the same annotation format as YOLO v5, and data produced with this annotation format is popular. Thus, it is safe to import the dataset as YOLO v5. After importing the dataset, the following code snippet exports the dataset to COCO format and stores it in the **`_annotations.coco.json`** file:

```
# export
coco_file = os.path.join(coco_dir, "_annotations.coco.json")
# Detectron requires starting index from 1
dataset.export.ExportToCoco(coco_file, cat_id_index=1)
```

Notably, Detectron2 requires the label index to start with index 1. Thus, **`ExportToCoco`** sets **`cat_id_index = 1`**. After performing the conversion, the YOLO annotation files under **`coco_dir`** should be removed using this snippet:

```
# now delete yolo annotations in the coco set
for f in txt_files:
  os.remove(f.replace(annotations_path, coco_dir))
```

Now, the conversion is completed. You can use the following snippet to view the structure of the generated dataset in COCO format:

```
!tree coco
```

The following is the output of this converted dataset:

```
coco
└ test
    ├ 00000_102.jpg
    ├ 00003_154.jpg
    └ _annotations.coco.json
```

Congratulations! By this time, you should have your dataset in COCO format, and it should be ready to be used by Detectron2. However, if your dataset is in Pascal VOC annotation format, the following section helps to perform this conversion.

Converting Pascal VOC datasets to COCO datasets

The Pascal VOC annotation format is another popular one. So, this section provides code snippets to convert it to the COCO annotation format before registering it with Detectron2. This section uses the simple dataset described previously as a demo (**`annotationdemo`**). First, we download the brain tumors dataset in Pascal VOC (**`voc.zip`**) and extract it on Google Colab:

```
# Download the dataset and unzip it
!wget url_to_voc.zip
```

```
!unzip voc.zip
```

With the dataset extracted, the following code snippet installs the **`tree`** package and views the structure of this dataset:

```
!sudo apt-get install tree
!tree voc
```

The following is the structure of the dataset in COCO annotation format:

```
voc
└ test
    ├ 00000_102.jpg
    ├ 00000_102.xml
    ├ 00003_154.jpg
    └ 00003_154.xml
```

Notably, the Pascal VOC data format stores one annotation file per image file with the same name and in a **`.xml`** extension.

This dataset can be modified to be in COCO format in place. However, the following code snippet copies these images and annotations into a new directory (**`coco_dir`**) for the COCO dataset and avoids modifying the original one:

```
import os
import shutil
from glob import glob
from tqdm import tqdm
```

```
voc_dir = "voc/test"
coco_dir = 'coco/test'
os.makedirs(coco_dir, exist_ok=True)
xml_files = glob(os.path.join(voc_dir, "*.xml"))
img_files = glob(os.path.join(voc_dir, "*.jpg"))
# copy annotations
for f in tqdm(xml_files):
  shutil.copy(f, coco_dir)
# copy images
for f in tqdm(img_files):
  shutil.copy(f, coco_dir)
```

Once copied, it is relatively easy to load this dataset as a **`pylabel`** dataset using the following code snippet:

```
from pylabel import importer
# load dataset
dataset = importer.ImportVOC(coco_dir, name="brain tumors")
```

Similarly, it is relatively easy to export the loaded dataset to a COCO style and store it in the **`_annotations.coco.json`** file with the following code snippet:

```
# export
coco_file = os.path.join(coco_dir, "_annotations.coco.json")
# Detectron requires starting index from 1
dataset.export.ExportToCoco(coco_file, cat_id_index=1)
```

Notably, Detectron2 requires the label index (**`cat_id_index`**) to start from **`1`**. Therefore, the **`ExportToCoco`** method sets this value to **`1`**.

Once the conversion is completed, it is safe to delete the Pascal VOC annotation files (`.xml` files) in the COCO dataset (`coco_dir`) using the following code snippet:

```
# now delete yolo annotations in the coco set
for f in xml_files:
  os.remove(f.replace(voc_dir, coco_dir))
```

Optionally, you can use the following statement to view the structure of the resulting dataset:

```
!tree coco
```

The following is the structure of the dataset in COCO annotation format:

```
coco
└ test
    ├ 00000_102.jpg
    ├ 00003_154.jpg
    └ _annotations.coco.json
```

Congratulations! By now, you should be able to convert datasets in Pascal VOC into COCO datasets that are ready to be consumed by Detectron2. If you have datasets in different formats, the steps to perform conversions should be similar.

Summary

This chapter discussed the popular data sources for the computer vision community. These data sources often have pre-trained models that help you quickly build computer vision applications. We also learned about the common places to download computer vision datasets. If no datasets exist for a specific computer vision task, this chapter also helped you get images by downloading them from the internet and select a tool for labeling the downloaded images. Furthermore, the computer vision field is developing rapidly, and many different annotation formats are available. Therefore, this chapter also covered popular data formats and the steps to convert these formats into the format supported by Detectron2.

By this time, you should have your dataset ready. The next chapter discusses the architecture of Detectron2 with details regarding the backbone networks and how to select one for an object detection task before training an object detection model using Detectron2.