CAL STATE FULLERTON
Department of Computer Science

This project has been satisfactorily demonstrated and is of suitable form.

This project report is acceptable in partial completion of the requirements for the Master of Science degree in Computer Science.

Recognizing Basketball Courts in Satellite Imagery Using the Detectron2 Framework
Project Title (type)

Jacob Re
Student Name (type)

Dr. Kanika Sood
Advisor's Name (type)

Advisor's signature Date

Please note that the reviewer's signatures does not appear here. I tried my best to get in contact with him towards the end of the semester so he could do a final review, but I think he was busy, as he never got back to me.

Dr. Kenytt Avery
Reviewer's Name (type)

Reviewer's signature Date

Abstract

Basketball is a very popular sport all across the world, and one of the ways in which some people enjoy getting exercise and/or interacting with others in their community is by going to local parks or other areas to play basketball. While there already exist tools to find basketball courts near a user's location (https://letsgoball.net/ and https://www.courtsoftheworld.com/), these tools are dependent on external users adding courts to these databases. This inspired the original goal of this project – to create an app or website similar to the tools above, but have the courts be auto located using a trained model to recognize courts from satellite imagery. While this initial inspiration was too large an undertaking for one person, the base need of training a model capable of recognizing basketball courts seemed much more feasible. Thus, the goal of this project was to create a model in the detectron2 framework that can predict on whether or not a satellite image contains a basketball court. Multiple different libraries (TorchGeo and rastervision) were considered, but detectron2 was settled upon due to its simplicity and lack of confounding needs for georeferenced data. A custom dataset with labels in the COCOjson format was created for the purpose of training this model – some images were custom labeled and processed, but the majority came from the NWPU VHR-10 dataset and its labels. This process of dataset creation proved to be the most challenging and time-consuming part of the project. Unfortunately, the trained model's final predictions were not as accurate as desired, likely due to the small dataset size and/or the lack of data augmentation; certainly, this accuracy could be increased with both a larger dataset and with data augmentation. The best predictions made by the model came with a final loss of .3453, but if this model were being used as the basis for an app or website similar to https://letsgoball.net/ and https://www.courtsoftheworld.com/, the accuracy would need to be much higher in order to return realistically accurate results. While the model resulting from this project may not be overly accurate, it still does provide a good starting point for future work in this area. For example, the model's accuracy could be increased by adding to the dataset and/or by implementing data augmentation. After a better model is obtained, it could then be possible to use said model as the basis for an app for website that can auto-detect basketball courts near a specified location, thus moving closer to the original goal of the project.

Table of Contents

Introduction

Background

One of the ways in which some people enjoy getting exercise and/or interacting with others in their community is by going to local parks or other areas to play pickup games and shoot around at basketball courts. As such, some people may wish for ways to find courts near them so they can discover places to play, especially if they do not have access to a basketball hoop at home. Naturally, there are some tools that already exist for such purposes, such as https://letsgoball.net/ and https://www.courtsoftheworld.com/, both of which have a somewhat standard functionality where users can enter their location or auto-locate to find courts in a specified radius. These tools include filters for the type of court, as well as ratings and reviews left by users for courts in the system. However, there are some limitations to these tools, and these limitations are what inspired this project.

Some limitations of these two tools include the fact that courts will appear only if users identify them and add them to the sites, which can result in courts that are missing from the databases, often being smaller courts at parks. Thus, if smaller courts tend to be the ones missing from these websites, most of the courts on said sites would be larger court complexes with multiple courts; these larger complexes tend to be busier and are more often in use for pickup games or by other people. As a result, these courts are better for people interested in playing pickup games or being in a busier setting but may not be as enticing for individuals who are looking to just "shoot around" or are searching for a more casual setting.

As such, these current tools are useful for finding larger complexes and more well-known courts, but the need for users to ID and add these courts may result in some smaller courts going unnoticed and unlisted on these sites. As was also discussed, these larger courts listed on the websites tend to be more popular and crowded, which can be a deterrent for casual players who may want to just shoot around or simply play in a smaller, quieter setting. Finally, the fact that smaller courts may be missing from these sites means that users may travel further than required to go to a court if they rely on these sites to find courts near them. All these factors combined to form the initial motivation for this project, which was adapted over time to a more feasible scope, given the time.

Initial Project Motivation and Changes

The initial idea for this project was to build a front end such as an app or a website, similar to that of the tools previously discussed. This app or website would present an interface like that of the current tools, where users can enter a location (or auto-locate) and a distance radius and be returned with a list of courts inside that radius. However, this project's implementation would differ from the current tools in that the list of courts would be generated by a model trained to recognize basketball courts from satellite imagery. In essence, the interface would take in the location and radius inputs, scan the nearby location's satellite imagery, make predictions using the model, and return a list of courts nearest the user, along with a confidence rating of the certainty of the detected object being a basketball court. However, upon further consideration, it was determined that implementing a front end, finding a way to source and automate the retrieval of localized satellite data, training a model to an acceptable accuracy, and creating the custom dataset needed to train that model would be far too much work for one person to accomplish over a single semester. As such, the goal of the project was adjusted to make it more feasible in the time given.

After consideration, it was determined that a reasonable goal to work towards would be attempting to train a model that is capable of recognizing basketball courts in static satellite images and also working on creating/procuring relevant data in order to train the model. Depending on the quality of the model, this model could then potentially be used as the basis for future work in implementing a front-end design (as was the original goal). The framework in which to train and implement the model also changed several times due to various challenges that were encountered throughout the research project, but the framework that was settled on was Meta's detectron2 framework for object detection in images (prior attempted approaches are discussed further in the "Implementation / First and Second Approaches" portion below).

Project Goals and Objectives

After all the considerations above, the finalized goal of this project was: to engage in training a model capable of recognizing basketball courts in satellite imagery using the detectron2 framework and to engage in the creation of a custom dataset made up of satellite

images of basketball courts and labels for said courts. The final deliverables of the project will consist of the following:

- A detectron2 model that can make predictions on recognizing basketball courts in static satellite images.
- A dataset that was used to train and test the model's accuracy, consisting of:
    - Training data used to train the model, consisting of images in .tiff and .jpg format and labels in a single .json file, formatted using the COCOjson format.
    - Test data (again, images in .tiff or .jpg format) used to try out the accuracy of the model and to demonstrate the sort of features the model is capable of recognizing.
- The code of the model itself and other code used to visualize and validate image labels.
- The code used for processing raw image labels into labels in the COCOjson format.
- A video demonstration of how the dataset is organized with its labels, how the model is set up and trained, and how the model makes predictions on test images.

In addition to these formal project deliverables, the other objective of this project is learning about object detection, dataset creation/management, image labeling, and about using the detectron2 framework. As will be discussed in the implementation section, the dataset creation and image labeling proved to be a very eye-opening process but was also quite tedious. As such, this proved to be a quality learning experience on satellite data in different forms. Additionally, it is possible that, given the model's accuracy, it could serve as the basis for a basketball locator application or website, as in the initial inspiration for this project. Unfortunately, as discussed in the results section, however, the model did not achieve a desirable accuracy level.

Related Work

As discussed in the Background portion, there are two tools that perform similar functions to the model being built in this project - those being https://letsgoball.net/ and https://www.courtsoftheworld.com/.

Development and Operational Environment

The final implementation of this project was developed primarily using the detectron2 framework developed by META. Programming was done locally in Visual Studio Code and in

Google Colab, and files were executed both locally and on the cloud, depending on the purpose of the program. The detectron2 framework is a PyTorch-based library, and as its GitHub page describes, it is “Facebook AI Research's next generation library that provides state-of-the-art detection and segmentation algorithms” [1]. The framework provides a multitude of pre-trained models and baseline results in their model zoo, which can be useful in serving as “backbones” to other custom trained models. Additionally, detectron2 allows users to define and train on custom datasets, which was perfect for the purpose of this project. One of the main downsides of this framework, however, is that it only has support for GPU training with NVIDIA CUDA GPUs. Because of this, the project was unable to be run locally to make use of the development machine’s Mac M1 GPU when training; this is the reason the model is defined and trained in Google Colab.

A second important tool that was used in the development of this project was the Desktop version of Google Earth Pro. Google Earth Pro allows users to download satellite imagery from the scene that is currently in the program window - users can change the resolution, presence of labels/place names, and other items when downloading the images. These images downloaded from Google Earth Pro serve as the secondary source of training data and the primary source of test data for this project. Additionally, Google Earth Pro allows users to draw polygons and other annotation shapes on the map and then export these polygons to .kml files to be used externally - this polygon feature was used in one of the initial image labeling attempts and will be discussed more later.

Finally, the third tool that was used for this project was the LabelMe application. This python application is a graphical image annotation tool that provides users with a simple GUI in which images can be labeled and annotated using polygons, rectangles, circles, lines and points [2]. These polygons can be labeled with a class name, and they can also be saved to a json file that contains other information about the image label that was just created. As such, the raw polygons generated by LabelMe were used to provide the saved Google Earth Pro imagery with labels that were semi-compatible with the detectron2 model.

Software Development Model

Incremental Model

The software development model that was used for this project was the Incremental Model. This model was chosen so that progress could be gauged based on the capabilities of smaller bits of the project. That is, the repeated style of the Incremental model was ideal for this project due to the ability to frequently observe and test the current deliverable product. This also allowed for the ability to change course easily, which ended up being necessary throughout the course of this project. Additionally, there were several different aspects of this project that needed to be handled individually before moving on to later tasks; the incremental model was ideal for this, as it allows for the building of small pieces of the project rather than large pieces all at once.

Requirement Analysis

As was just discussed, the incremental model allows for the building of small portions of the project rather than larger, more complete pieces. Thus, this idea of "smaller portions" also applies to the requirements analysis portion of this model. That is, for each of these smaller, incremental pieces, requirements needed to be gathered in order to determine what the end product or goal should be. For this project, requirements analysis was not overly extensive, but there were still some high-level areas that were determined to be necessary to elicit requirements for. These areas include: the dataset used to train the model, labels for the images in the dataset, and the trained model that is able to make predictions on unseen imagery.

The dataset used to train the model needed to be a set of static satellite images, most of which contained basketball courts, and some of which did not (this was done in order to expand the variety of scenes the model was exposed to in training). These images also need to have labels associated with them so that the model was able to determine the features it needs to pay attention to while training. Images in the dataset were pulled from two sources: the Northwestern Polytechnical University very-high-resolution ten-class remote sensing image dataset (NWPU VHR-10), and from custom images downloaded from Google Earth Pro.

The labels used to specify the locations of the basketball courts in the images needed to be contained in a single file, as that is the format the detectron2 framework is expecting. The file

must be a .json file that follows the COCOjson format [3]; this format will be discussed more later. The result of training on the dataset is a .pth file that can be used to make predictions on images obtained in a way similar to that of the training data. This .pth is the trained model weights, and must be loaded into the program in order to make predictions.

Design and Development

There were several different "Design and Development" portions that were undertaken for different parts of the project; all of these were necessary, as they focus on specific, important portions of the project.

The first of these portions was developing a base model that would be used as the starting point for the basketball court recognition model. The second of these portions was finding data that could be used to train the model. Data for this project was sourced from two locations: firstly, the NWPU VHR-10 dataset provided labeled images (in COCOjson format) for 151 courts from 86 files. Secondly, custom images downloaded from Google Earth Pro, along with manually added labels, contributed images of 25 courts from 11 files. This brought the total number of courts the model was trained on to 176 courts from 97 image files. Adding in negative images that did not contain basketball courts, the total number of image files the model was trained on was 99. The final of these portions was training the model on the custom dataset and also developing a way to have the trained model make predictions and then visualize said predictions.

Testing

The testing portion focused on validating the functionality of each of the portions from the previous "Design and Development" section. Testing that was necessary for setting up the base model included: verifying the installation of PyTorch, verifying installation of detectron2 from its GitHub page, verifying the mounting of the Colab Notebook to Google Drive (this is where the dataset is stored), verifying that the model could read image labels and then display each image with its corresponding label, verifying that the model was capable of fully running, training, and producing an output, and finally, verifying that the trained model could be used to attempt to make predictions on different images.

While testing the dataset consisted of only a few steps, these steps proved to be quite difficult and time consuming due to the large quantity of images in the dataset. Testing in this phase mainly consisted of verifying that the image labels were in a format recognized by the detectron2 framework. The detectron2 framework expects, in a custom dataset implementation, a folder containing training images, and then a single .json file (in COCOjson format) containing all the labels for these images. A folder of similar structure with test images can also be passed to the model, but this is not necessary, as it is quite easy to create a custom implementation that uses the trained model to predict on images - which was done for this project. A folder containing validation images can also be specified, however, the detectron2 framework does not have an out-of-the-box way to add and use validation datasets with its models. Creating a validation dataset and implementing and debugging the validation dataset proved to be too time consuming for the course of this project. Thankfully, the images used as inputs simply needed to be in a standard image format (.tiff, .png. .jpeg, etc.), so this step did not require extensive testing to verify.

Implementation

Just as the testing portion focused on validating the functionality of each of the portions from the Development & Design section, the implementation portion focused on how each of the portions previously discussed were implemented. The base model for this project was designed using a detectron2 tutorial from the official documentation that gave an example of how to set up, train, and use a model for object detection in images while pulling from a custom dataset [4]. This base model makes use of pre-trained models from the detectron2; these models can be used as "backbones" for custom user-trained models. For this project, the "**`COCO-InstanceSegmentation/mask_rcnn_R_50_FPN_3x.yaml`**" model was used as the basis.

As described, creating, labeling, and formatting the custom dataset proved to be the most time-consuming aspect of the project; however, it was also very necessary, as it would be extremely difficult to train a model without an appropriate dataset. The images for the dataset were downloaded from Google Earth Pro, and consisted of basketball courts of varying colors, sizes, locations, orientation angles, court line designs, etc. that were saved in .tiff format for best image quality. In addition to images containing basketball courts, a small number of images

without basketball courts were also saved and added to the dataset. The images containing basketball courts were then labeled using the LabelMe tool, in which polygons were drawn around the boundary of the basketball court(s) in the image. These polygons were then saved to a .json file, one for each image, after which their data was passed through a function which partially converted the data to COCOjson format. This data still needed slightly more manual processing to be fully correctly formatted, but after this processing, the data for the polygon was added to the dataset label file. This project implemented a simple program (**`verify_labels.py`**) used to verify the placement of the labels on the images; users can use the command line to specify an image file to display the labels of (if labels exist for that image).

The final portion - training the model on the dataset and using the model to make and visualize predictions - proved to be much easier than creating the dataset. The model was trained following the standard detectron2 workflow, and the progress of the model's training progress was tracked using the output to the command line. To assist with predicting labels and visualizing those predictions, this project implemented a simple custom program to allow users to visualize predictions made by the model; this is done by specifying an image file in the command line along with the **`detectron2_model_predict.py`** file.

Software Requirement Specification

Overall Description

Software Features

a. Semi-Automated Adding of New Data to Database: The code contained in this project allows for any user to add additional labeled images to the dataset by carrying out the same steps as described in the Implementation portion of this report. While this process is not nearly fully automated, it is still made significantly easier by the helper programs.
b. Model Predictions: A user of this code can use the trained model (the .pth file is included in the submission along with this project) to make predictions on images. These images can come from the Test folder contained in the custom dataset or from custom images procured by the user.
c. Confidence Level Display: When making predictions, the detectron2 framework provides a confidence rating about the object in the scene that is being predicted on. These

confidence ratings (displayed as percentages) are shown when displaying predictions made by the model.

d. Displaying Image Labels Over Images: Users are able to use one of the helper programs included with this project (**`verify_labels.py`**) to display an image and its corresponding label(s) in order to verify the placement of the label. However, the helper function is only able to display images and labels that are contained in the detectronDataset folder.

User Classes

a. Regular Users
    i. This class could be any user with a relatively strong knowledge of programming. A regular user would need a knowledge of programming in order to make use of some of the helper programs, such as the ones that make predictions and display labels. These programs need to be run from the command line (and also need to have file paths their code modified) in order to function properly.

b. Expert Users
    i. This class of users really only contains one person – the person who designed this project and wrote all of the code. The small size of this user class is due to the fact that many of the file paths and much of the file access in these programs are hard-coded into the text of the program file, and are thus not able to be dynamically changed. As such, it would be difficult for anyone besides the author of this project to do much with the code beyond basic actions, such as displaying labels, making predictions, and creating and formatting new images and labels.

Operational Environment

a. This project shall run on Windows, MacOS, and Linux environments, so long as the user is able to install PyTorch and detectron2.

b. The model training portion of this project was run on Google Colab for the purposes of GPU access, but this model training could also be done locally if the user has access to an NVIDIA GPU.

Design Implementation Constraints

1. The machine/environment that runs this code must be capable of installing and running the latest versions of PyTorch and detectron2.

2. Model training can only be done locally with a GPU if the local machine has access to an NVIDIA GPU, as the detectron2 framework only supports being run on NVIDIA CUDA GPUs.

Functional Requirements

- FR – 01: The programs SHALL allow users to add images and labels to the detectronDataset in a semi-automated fashion.
    - Description: The programs SHALL allow users to make use of the helper programs and the processes listed in the Implementation portion in order to obtain new satellite imagery, add labels to it, convert these labels to COCOjson format, and then add these images and labels to the dataset
    - Pre-Condition: User has access to various program files (these are: **`jsonCOCO_processing.py`** and **`finalCOCO_dataprocessing.py`**).
    - Post-Condition: User has added the image file and its appropriate label data to the dataset and its label file.
    - Dependencies: User must have the LabelMe tool downloaded, as this is what is used to label the images files, and its output format is what the files listed above are expecting.
    - Risks: It is quite easy to accidentally make a mistake when transferring the data over from the labels to the dataset, and issues in the dataset can cause issues if a new attempt to train the model is made.
- FR – 02: The programs SHALL allow users to view image files and their corresponding labels in order to verify the placement of said labels.
    - Description: The programs SHALL allow users to make use of a helper function that will enable the user to visualize an image and its corresponding labels, ensuring that the labels are correctly positioned.
    - Pre-condition: User has access to various program files (in this case, all that is needed is **`verify_labels.py`**).
    - Post-Condition: User is presented with a window containing the image and its label, from which the user can verify the placement of the label(s).
    - Dependencies: The image file must be in the detectronDataset folder with the labels in the label file in this same folder. Additionally, the code in the

`verify_labels.py` program must be modified to ensure the correct file path is being referenced.

- Risks: None

- FR – 03: The programs SHALL allow users to use the included trained model to make predictions on image files originating from the Test folder of the dataset or from other sources.
  - Description: The program SHALL allow users to make use of the .pth file (the trained model weights) in order to make predictions on whether or not a certain image file contains a basketball court, and where in the image that potential basketball court is located. These images can be the images contained in the Test folder of the detectronDataset or can be images obtained from a different source that have been added to that Test folder.
  - Pre-Condition: User has access to various program files (in this case, all that is needed is **`detectron2_model_predict.py`**) as well as images on which to predict.
  - Post-Condition: An image containing the model's predictions and its certainty level are displayed to the screen.
  - Dependencies: The image file that is being predicted on must be in the Test subfolder of the detectronDataset folder, as this is where the program's file path expects the image to be located. Additionally, the code in the **`detectron2_model_predict.py`** program must be modified to ensure the correct file path is being referenced.
  - Risks: None

External Interface Requirements

User Interfaces

a. UI – 1: Images displayed by various program files (i.e., those that verify the image labels or display the predictions) SHALL be displayed in regular program windows for the user to view.

b. UI – 2: All user inputs of file names SHALL be inputs to the command line after the program has begun running.

Hardware Interfaces

a. HI – 1: The program files SHALL be able to be run on desktop/laptop computers running Windows, Mac OS, or Linux that are able to run python scripts.

Software Interfaces

a. SI – 1: Images labels SHALL be stored in a single .json file adhering to COCOjson formatting rules, and image files shall be stored in the same folder as their labels. The image files must be in a standard image file format.

Communications Interfaces

a. None

Non-Functional Requirements

Performance Requirements

a. None

Security Requirements

a. None

Software Quality Requirements

a. SQA – 1: The model's predictions SHALL provide a certainty rating that describes how certain of a prediction the model has made.

Context Diagram

Figure A was supposed to represent the Context Diagram, but I ran out of time
Fig A Here

The context diagram above (Fig. A) gives a view of the project as a single process (the process of adding custom images to the dataset has been simplified here). This model describes a typical flow of data through this project.

Control Flow Diagram

The Control Flow Diagram of this project is included below in Fig. B.

Figure B was supposed to represent the Control Flow Diagram, but I ran out of time
Fig B Here

Use Cases

1. UC – 01: Adding custom images and labels to the detectronDataset.
    a. Description: Users are allowed to procure their own satellite imagery, create their own labels, and add said data to the custom dataset. This can be done in a semi-automated fashion using the helper programs included with this project.
    b. Actor(s): Regular and expert users
    c. Pre-condition(s): The image/labels must not already exist in the dataset, and the images must be labeled properly and in the correct format.
    d. Scenario:
        i. The user finds a satellite area they wish to include as an image and downloads this satellite imagery using Google Earth Pro.
        ii. The user labels the correct area of the image using the LabelMe tool.
        iii. The helper programs are used to convert these labels to be in the correct format, and the data is copied to the label file.
    e. Post-condition: The image and its label are added to the dataset and label file.
2. UC – 02: Verifying the placement of labels over dataset images.
    a. Description: Users can verify the location of image labels using the **`verify_labels.py`** program. This helps ensure that each label (especially any custom labels) is properly positioned on the image.
    b. Actor(s): Regular and expert users.
    c. Pre-Condition(s): The image file inputted by the user must be contained in the detectronDataset folder and this image must at least be referenced in the label file's **`images`** portion.
    d. Scenario:
        i. The user runs the **`verify_labels.py`** file from the command line, and then enters the name of the image file for which they want to verify the labels.
        ii. If the image file has any label(s), these labels will be displayed, along with the image, in a window for the user to inspect.
    e. Post-Condition(s):
        i. The location of the image label for the file in question has be verified.

3. UC – 03: Using the trained model to make predictions on novel images.
    a. Description: Users can use the **`detectron2_model_predict.py`** file and the included .pth file (the trained model weights) to make predictions on images about whether or not those images contain basketball courts.
    b. Actor(s): Regular and expert users
    c. Pre-Condition(s): The image file specified by the user must be contained in the detectronDataset folder.
    d. Scenario:
        i. The user has an image that they want to use the model to make a prediction on. The user first adds their image file to the detectronDataset folder.
        ii. The user runs the program from the command line and specifies the image file name that was just added.
        iii. The model makes predictions on the image and displays these predictions, along with a confidence rating, to the user in a window for inspection.
    e. Post-Condition(s):
        i. The model has made a prediction about the contents of the image specified by the user.

Use Case Diagram

Below is the Use Case Diagram for this project (Fig. C).

Figure C was supposed to represent the Use Case Diagram Diagram, but I ran out of time
Fig C Here

Activity Diagram

There are three activity diagrams included with this report, one for each use case. Fig. D depicts UC – 01, Fig. E depicts UC – 02, and Fig. F depicts UC – 03.

Figure D was supposed to represent the Activity Diagram for UC – 01, but I ran out of time
Fig D Here

Figure E was supposed to represent the Activity Diagram for UC – 02, but I ran out of time
Fig E Here

Figure F was supposed to represent the Activity Diagram for UC – 03, but I ran out of time
Fig F Here

Design and Architecture

Architectural Diagram

Figure G was supposed to represent the Architectural Diagram, but I ran out of time
Fig G Here

Class Diagram

Figure H was supposed to represent the Class Diagram, but I ran out of time
Fig H Here

System Sequence Diagram

A sequence diagram in Figure 9 shows how objects interact with each other. The following diagram is sequence diagram for the application.

Figure I was supposed to represent the Class Diagram, but I ran out of time
Fig I Here

Implementation

First and Second Approaches

The first attempted approach, as was suggested by the reviewer for this project, was to use the TorchGeo library to train the model. TorchGeo is "PyTorch domain library, similar to torchvision, providing datasets, samplers, transforms, and pre-trained models specific to geospatial data." [5]. At first, this seemed like the perfect library for the purposes of this project. However, upon exploring the library more, it proved to be quite complex and difficult to understand. Additionally, the library is a somewhat new release, meaning that there were really no basic setup tutorials or example models in the TorchGeo documentation, which made it quite difficult to set up a base model to use as the starting point for the trained model. This, combined with the unfamiliar syntax of TorchGeo, led to the first change in approach - from TorchGeo to rastervision.

The TorchGeo documentation page has a list of similar tools, and that is where rastervision was found. Rastervision is "an open-source library and framework for Python developers building computer vision models on satellite, aerial, and other large imagery sets" [6]. Similarly to TorchGeo, the rastervision framework also provides datasets, samplers, and pre-trained models specific to the library. As mentioned, rastervision (2018) [6] is an older framework than TorchGeo (2021) [5], meaning there are more tutorials, answered questions on discussion boards, and other general resources associated with the rastervision framework, as it has been in use for longer.

Because the rastervision library is quite similar in purpose to TorchGeo and because it had more resources, it at first seemed like the library would also be a good fit for this project. However, as the project progressed, it became clear that rastervision, or any library designed specifically for geospatial data for that matter, might not be the best fit for this project. The main problem that was encountered during the rastervision attempt came with the labeling of imagery.

In this approach, images were still downloaded from Google Earth Pro, but there were two different attempts to label them. The first was a brute force method where the image was loaded into a window by the matplotlib library, and then the cursor was used to move to the points that represented a polygon around the basketball court. The matplotlib window gives the cursor coordinates in the image, so these coordinates were written down, and then this process was repeated to draw a full polygon (really just a rectangle) around the court in the image. These

resulting points were saved to a geoJSON file; geoJSON is “a format for encoding a variety of geographic data structures” [7]. geoJSON files support a wide variety of different label shapes, such as Point, LineString, Polygon, MultiPoint, MultiLineString, and MultiPolygon. Initially, it appeared as though this approach at labeling the images was working quite well, as as Fig. 1 below demonstrates.

Figure 1 – Displaying GEOjson polygons over satellite imagery

However, when these images and their labels were passed into the base rastervision model and visualized there, it became clear that these labels were not recognizable by the model. When visualized using the rastervision capabilities, the labels did not appear on the image as they did in Fig. 1 and Fig. 2. Upon doing some research, one possible issue that was found was that these polygon labels were calculated relative to the pixels in the image, and not relative to the satellite coordinates of that actual location on the earth. This is because these geospatial learning libraries are usually expecting coordinates that are relative to [latitude, longitude] style. This was confirmed upon the examination of the label file included with the example rastervision model, as these labels were in a [latitude, longitude] format.

Upon this discovery, an alternative approach to labeling the satellite imagery was taken. Instead of the manual, brute force approach described above, the built in Google Earth Pro “Draw Polygon” feature was used, as the coordinates recorded by this method are represented in geographical [latitude, longitude] format. These polygons were initially saved as .kml files,

which were then converted to geoJSON format using the ASPOSE free online KML to geoJSON converter [10]. However, upon visualizing these new labels in the same way using the rastervision framework, it once again became clear that these labels were not correct either, as the labels still did not appear. While it is not certain what exactly caused these geoJSON labels to function incorrectly, it likely had something to do with the fact that the images being used did not contain georeferencing data, which is necessary in order to place the coordinates in the correct area on the globe.

At this point, it became clear that attempting to work with geographic-based frameworks was introducing too many confounding aspects to this project, so the approach being taken towards the project was re-examined. Upon consideration, it was determined that, at a basic level, this project can be classified as an object detection in imagery program. That is, the georeferencing data and latitude and longitude coordinates are irrelevant to the base purpose of the project – training a model that can make predictions on whether or not an image contains a basketball court. This idea led to the finalized approach that this project took.

Finalized Approach

As mentioned previously in the "Development and Operational Environment" section, the library that was settled on for this project was the detectron2 framework. This framework proved to be much more easily usable than the prior two attempted libraries, and the removal of geospatial data/coordinates from the process made things significantly easier. As such, the finalized approach to this project made use of the detectron2 framework, Google Earth Pro for some image sourcing for the training data, the NWPU VHR-10 dataset, the LabelMe program for labeling captured images, and the COCOjson format for object detection and segmentation.

Dataset Construction

As has been discussed, the creation of the custom dataset needed to train a model such as this proved to be the most difficult and time-consuming part of this project. Partially, this was due to failed attempts to properly label the imagery being used. However, this process was also just tedious and required some level of manual intervention, no matter how automated the process became. The discovery of the NWPU VHR-10 dataset made things a lot easier,

thankfully, as this provided many more courts with labels in COCOjson format – all that needed to be done was to extract the appropriate labels and add the information to the .json label file in the custom dataset. For images in the dataset that were not sourced from the NWPU VHR-10 dataset, the collection, labeling, and verification process proceeded as follows.

Custom Dataset Entries

The first step was to find an area of satellite imagery to use as an image; usually, this image contained at least one basketball court, but this was not always the case. Courts visible via satellite were found partially through the author's personal knowledge of basketball courts in the Orange County, California area, and partially through using the related tools (https://letsgoball.net/ and https://www.courtsoftheworld.com/) to find existing outdoor basketball courts. Once a court was found, Google Earth Pro was used to navigate to the location, and the location was zoomed in on and moved around to accommodate the important

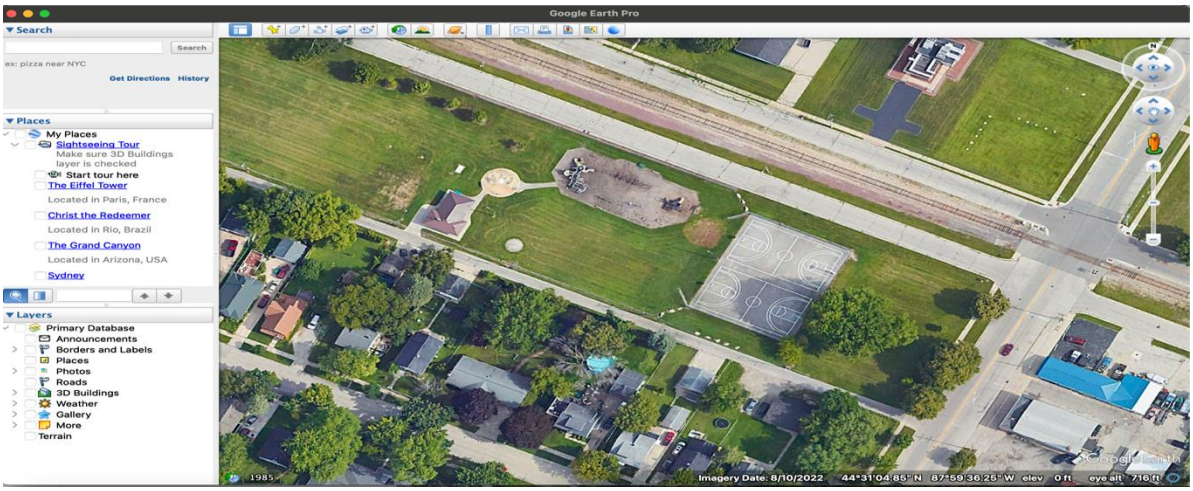


Figure 3 - View of a court in Google Earth Pro

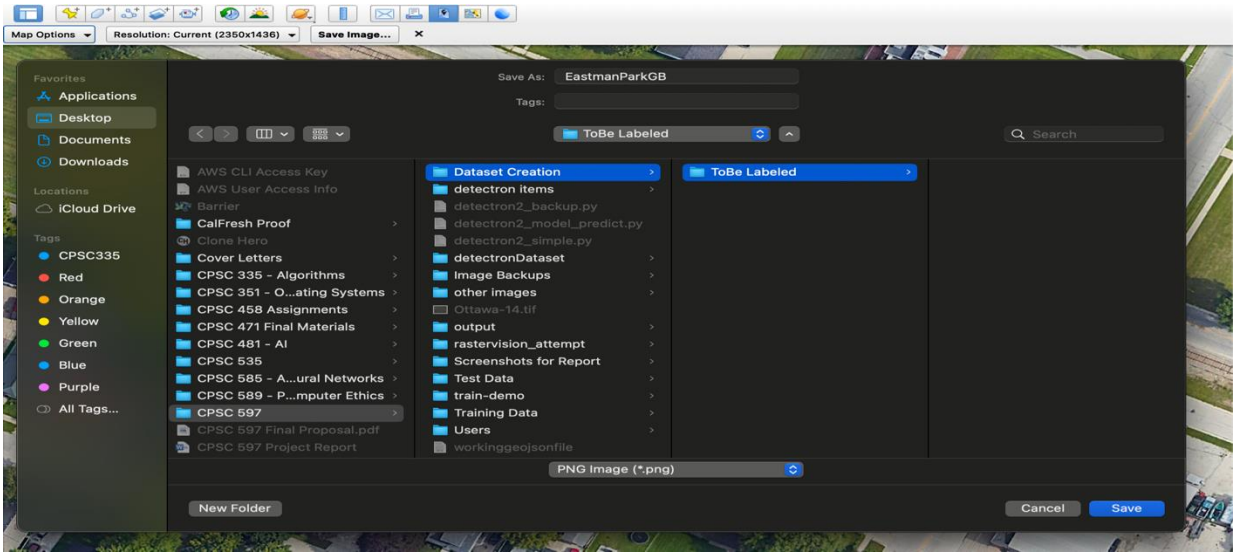


Figure 4 - Saving an image in Google Earth Pro

aspects of the scene (Fig. 3). After this, the image was saved to the local machine (Fig. 4) in a folder that designated files to be labeled. Upon the collection of various images containing basketball courts, the next step was to label these images so that the model would be capable of determining the features in the scene to focus on and learn on. Images were initially labeled using the LabelMe tool, as shown in Fig. 5 below. These labels were saved by LabelMe as coordinates in a .json file.

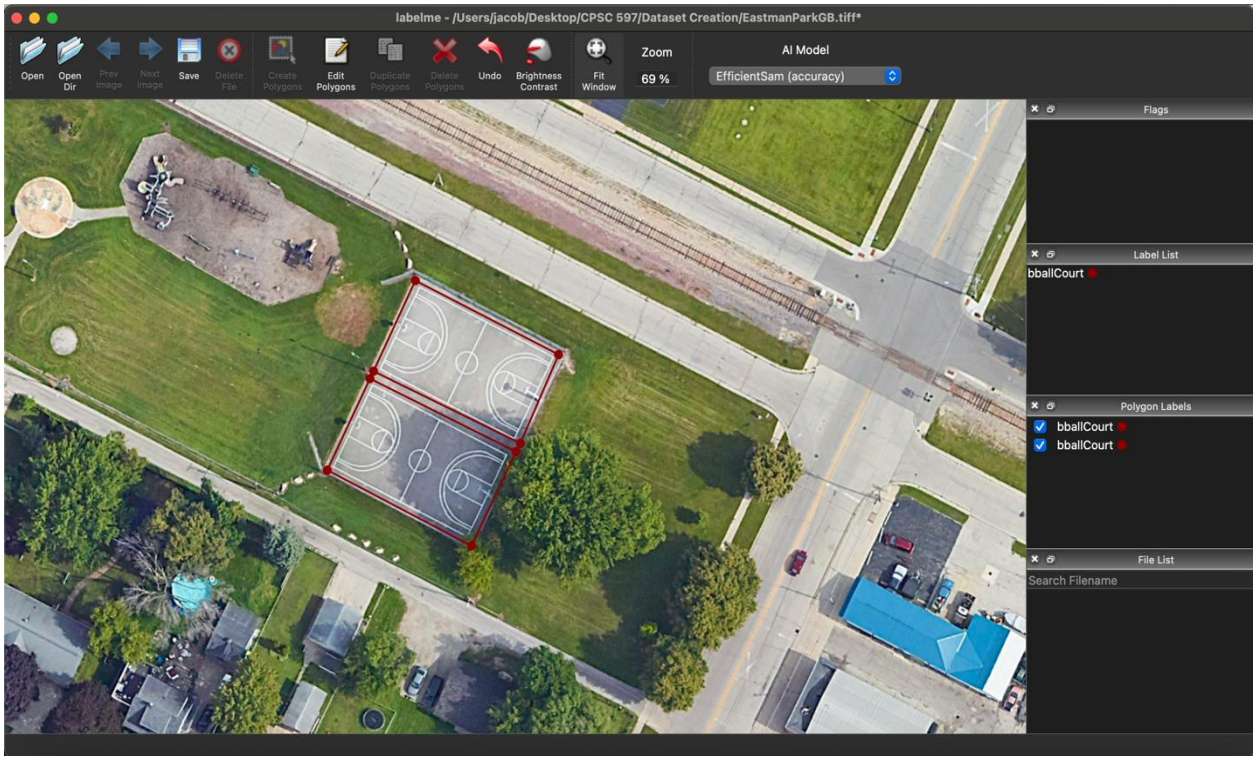
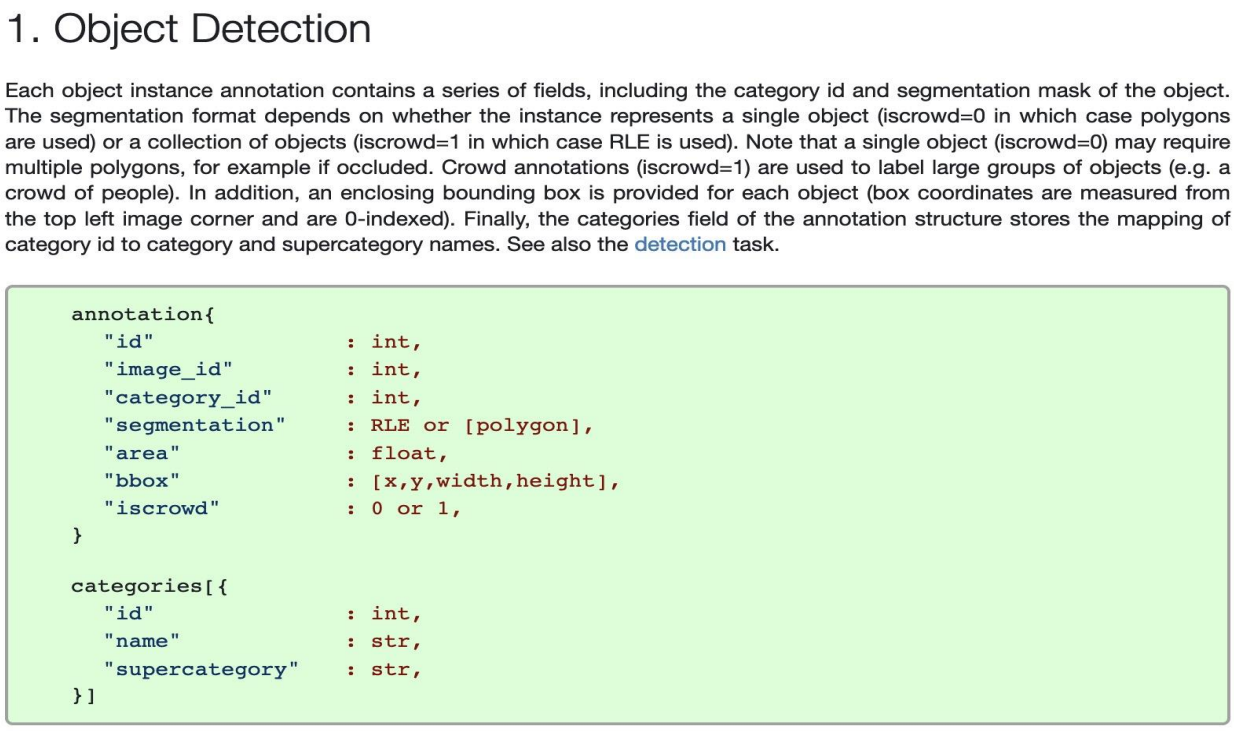


Figure 5 – Labeling saved image using LabelMe

Once images were labeled and the image labels were saved to the local machine, these labels needed to be processed in order to be in the COCOjson format (this is the format detectron2 expects). The COCOjson (COCO meaning Common Objects in Context) format is "a data format used for annotating and managing objects in images or videos," [8] thus helping to explain why the detectron2 framework expects the labels in this sort of file format.

The COCOjson format has different annotation styles for different purposes, such as for object detection, stuff segmentation, image captioning, etc., [9] and as this project focused on object detection in images, the object detection format is what was used to label the images. The COCOjson site provides a basic outline for the object detection format (Fig. 6), which was used when converting from the LabelMe polygons to the COCOjson polygons.

As the text in the figure (Fig. 6) describes, every time an annotation for an object is added, certain fields need to be specified in order to describe the object. First is the **`id`** field,



1. Object Detection

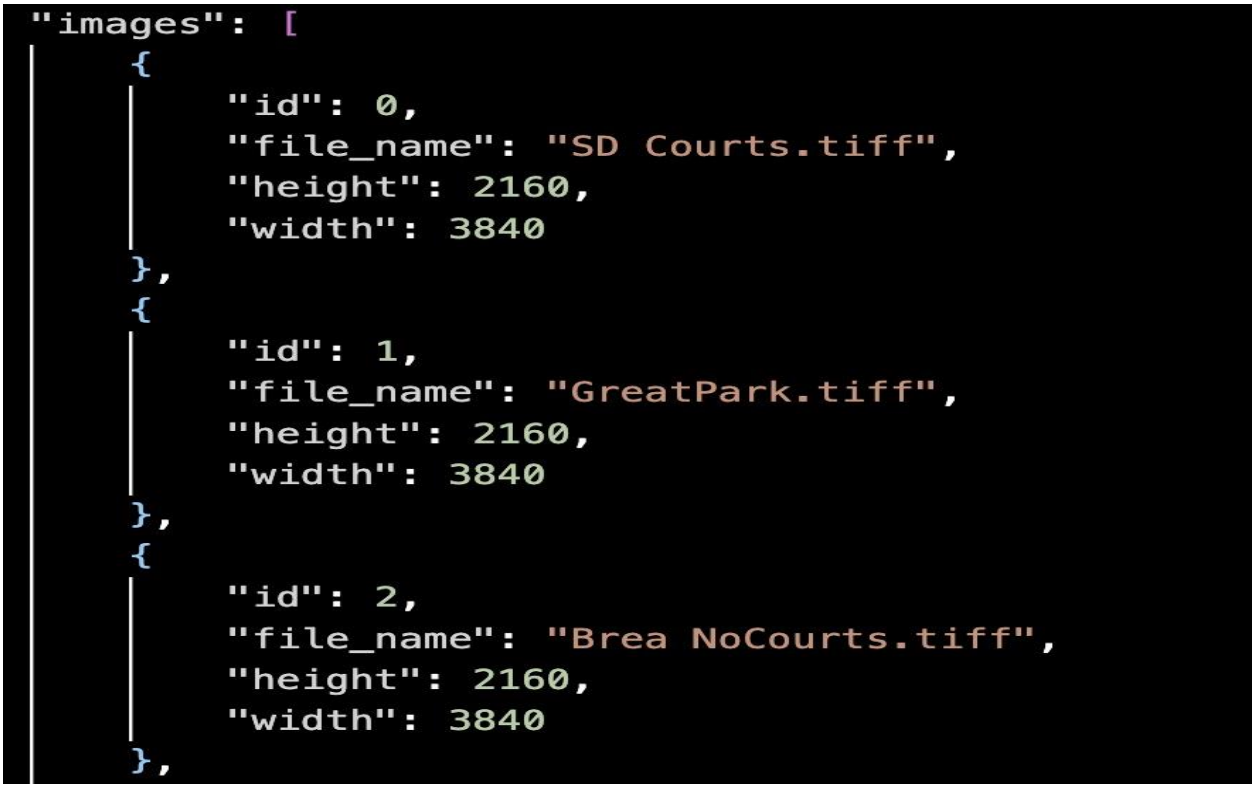
Each object instance annotation contains a series of fields, including the category id and segmentation mask of the object. The segmentation format depends on whether the instance represents a single object (iscrowd=0 in which case polygons are used) or a collection of objects (iscrowd=1 in which case RLE is used). Note that a single object (iscrowd=0) may require multiple polygons, for example if occluded. Crowd annotations (iscrowd=1) are used to label large groups of objects (e.g. a crowd of people). In addition, an enclosing bounding box is provided for each object (box coordinates are measured from the top left image corner and are 0-indexed). Finally, the categories field of the annotation structure stores the mapping of category id to category and supercategory names. See also the detection task.

```
annotation{
   "id"               : int,
   "image_id"         : int,
   "category_id"      : int,
   "segmentation"     : RLE or [polygon],
   "area"             : float,
   "bbox"             : [x,y,width,height],
   "iscrowd"          : 0 or 1,
}

categories[{
   "id"               : int,
   "name"             : str,
   "supercategory"    : str,
}]
```

Figure 6 – COCOjson basic format and description

which is simply the number of the annotation in the sequence in the .json file. That is, the first annotation in the file has an **`id`** of **`0`**, the second has an **`id`** of **`1`**, and so on. Next was the



```
"images": [
    {
        "id": 0,
        "file_name": "SD Courts.tiff",
        "height": 2160,
        "width": 3840
    },
    {
        "id": 1,
        "file_name": "GreatPark.tiff",
        "height": 2160,
        "width": 3840
    },
    {
        "id": 2,
        "file_name": "Brea NoCourts.tiff",
        "height": 2160,
        "width": 3840
    },
```


Figure 7 – **`images`** *portion of the COCOjson label file*

`image_id`, which determines which image file the from the images portion (Fig. 7) the annotation is associated with; specifically, the **`image_id`** in the **`annotations`** portion of each object (Fig. 6) is tied to the **`id`** of each image listing in the **`images`** portion (Fig 7). After this is the **`category_id`**, which determines the type object that the annotation is describing.

In datasets with a larger variety of labeled objects (such as the NWPU VHR-10 dataset), the **`category_id`** is used to differentiate between different types of objects, such as a boat, plane, baseball diamond, tennis court, etc. In this case, the categories portion assigns an **`id`**, **`name`**, and **`supercategory`** (optional) to each object. This **`id`** becomes the **`category_id`**, the name is just a simple description of the object, and the optional **`supercategory`** can be used to distinguish between different groupings of objects if there are a lot of objects described in the file. For example, "boat" and "plane" could belong to the **`supercategory`** "transport," while "baseball diamond" and "tennis court" could belong to "sports." In the case of this project, there was only one object being labeled (basketball courts), so the **`supercategory`** field was omitted, while the **`id`** was **`0`**, as there was only a single object being labeled.

After this came **segmentation**, for which the layout gives two options: **RLE** or **[polygon]**. As the description in Fig. 6 says, the choice of **RLE** vs **[polygon]** depends on the value of **iscrowd**, with **iscrowd == 1** using RLE and **iscrowd == 0** using **[polygon]**. **iscrowd == 1** represents a large group of objects (such as a crowd of people), while **iscrowd == 0** represents single objects being annotated. Thus, for this project, the **[polygon]** format was used, as the only objects being labeled were individual basketball courts (multiple courts in the same image call for multiple unique annotations, all with the same **image_id**).

The **[polygon]** format is very similar to that of the geoJSON files that were used in the initial approaches to this project. This object in COCOjson is represented by a list of points that, when connected, outline the shape of the object being labeled. This allows for labeling of objects with complex shapes, but for this project, only four sets of points were needed to outline most of the basketball courts, all basketball courts are rectangles. After this was the **area** portion, which is simply the area of the shape inside the polygon; a helper function, **polygonarea.py**, was created to calculate this area, and this will be discussed more later.

Next came the **bbox** field, which represents the bounding box around the object. As listed in Fig. 6, the bounding box is in the format **[x, y, width, height]**, where the x and y represent the point that is the top left corner of the bounding box, and the width and height are the width and height of the box. As a result of this format, the bounding boxes will always be rectangles that are square with the edges of the image, even if the actual object being labeled is at an angle in the image. Bounding boxes differ from polygons in that the bounding boxes can only be rectangles that are square with the screen edges, which doi not allow for more complex shapes or shapes at angles, whereas polygons allow for more detailed outlining of more types of shapes. However, these polygons and bounding boxes work together to describe the bounds and outline of the object that is being labeled.

Finally came the **iscrowd** field, which, as previously discussed, is set to 0 due to the nature of this project. In addition to these fields listed in Fig. 6, there were also two more added fields, one descriptive and one necessary. The first is that there was a **file_name** field added to each entry in the annotation portion; this was simply used to better associate the label with the proper image. The second is the **bbox_mode** field, which can be used to determine the format

in which the bbox values will be interpreted. For the case of this project, **`bbox_mode`** was set to 1, as this indicates that the bbox values will be in the format described by the COCOjson page (**`[x, y, width, height]`**).

After labeling the basketball courts with polygons in LabelMe (Fig. 5) and saving these labels, the next step was to convert data in the .json files generated by LabelMe to be in COCOjson format. Fig. 8 shows an example of the structure of the .json files before being converted to COCOjson format. As the figure shows, the only information provided is the coordinates of the points, the label name, and other non-important data below; this is far from all of the data that is needed to satisfy the COCOjson format. The first step of this conversion was to use the helper program, **`jsonCOCO_processing.py`**, to partially convert the data to COCOjson format. This helper program was adapted from code in the detectron2 Google Colab tutorial, specifically the portion that prepares the data to be in detectron2's expected format [3].


```
"shapes": [
  {
    "label": "bballCourt",
    "points": [
      [
        1457.608695652174,
        825.9710144927535
      ],
      [
        1519.9275362318842,
        686.8405797101449
      ],
      [
        1286.5942028985507,
        570.8985507246376
      ],
      [
        1218.4782608695652,
        712.927536231884
      ]
    ],
    "group_id": null,
    "description": "",
    "shape_type": "polygon",
```


Figure 8 - .json file structure before conversion to COCOjson format

As Fig 9 shows, the helper program will loop through each object in the **`shapes`** portion (each polygon) and will record various data about the image, such as the height, width, file path, etc. This code was adapted to read the format that LabelMe outputted for the

```
for idx, shape in enumerate(data["shapes"]):
    record = {}
    (variable) record: dict  the image and store in the "record" dictionary
    ["imagePath"]
    record["image_id"] = idx
    record["height"] = data["imageHeight"]
    record["width"] = data["imageWidth"]
    # Find the min and max x and y values of the shape drawn in LabelMe
    # These values will be used to fill out the bbox field
    min_x = min(x[0] for x in shape["points"])
    min_y = min(x[1] for x in shape["points"])
    max_x = max(x[0] for x in shape["points"])
    max_y = max(x[1] for x in shape["points"])

    # These values will be used to assign the values in the bbox field
    x = min_x
    y = min_y
    width = max_x - min_x
    height = max_y - min_y
    # Create an object that will be saved to the json file
    obj = {
        # Assign the values of the bbox field to their correct place
        "bbox": [x,
                 y,
                 width,
                 height],
        # Set bbox mode to 1 so that the bbox coordinates are interpreted correctly
        "bbox_mode": 1,
        # Passes the polygon points as the segmentation points
        "segmentation": [shape["points"]],
        # Use 0 for the category, as there is only one type of object being detected
        "category_id": 0,
```

Figure 9 – main portion of **`jsonCOCO_processing.py`**


```
"file_name": "../../../Dataset Creation/EastmanParkGB.tiff",
"image_id": 0,
"height": 1436,
"width": 2350,
"annotations": [
    {
        "bbox": [ 1218.4782608695652, 570.8985507246376, 301.449275362319, 255.07246376811588 ],
        "bbox_mode": 1,
        "segmentation": [
            [
                [ 1457.608695652174, 825.9710144927535 ],
                [ 1519.9275362318842, 686.8405797101449 ],
                [ 1286.5942028985507, 570.8985507246376 ],
                [ 1218.4782608695652, 712.927536231884 ]
            ]
        ],
        "category_id": 0
    }
```


Figure 10 – Result of passing the initial json file through helper program

labels, as these labels were in a different format than those read by the example code. After this, the minimum and maximum x and y values were extracted to calculate the values for the **`bbox`**

section. Then, an object was created to store all of the data that will be placed in the new annotation; this includes the **`bbox`** values, the **`bbox_mode`**, the segmentation points, and the **`category_id`**. This data is then added to the annotations portion of the output .json file, which is then saved to the local machine. Fig. 10 shows the structure of the file that resulted from passing the initial labels through the **`jsonCOCO_processing.py`** file.

While program does a decent job of converting the data into the COCOjson format, it still required some manual processing before it was fully in the COCOjson format. Specifically, the segmentation portion needed to be changed from a list containing four individual lists of points (as in Fig. 10) to a list containing eight individual points. For example, the segmentation data in Fig. 10 needs to become: **`[ 1457.60, 825.97, 1519.92, 686.84, … ]`**. In addition to formatting the **`segmentation`** data, the area of the polygon also needed to be calculated, as this was not provided by the LabelMe .json file. A few more fields needed to be added as well, and Fig. 11 shows an example of a finalized annotation label that the detectron2 framework will accept. The rest of this conversion (for the segmentation, area, and other fields) was done using another helper program, this one named **`finalCOCO_dataprocessing.py`**. While this program file is too large to include in Figures in this paper, the purposes and processes of this file are discussed below.


```
{
    "id": 3,
    "image_id": 0,
    "file_name": "SD Courts.tiff",
    "category_id": 0,
    "bbox": [ 636.2549800796812, 194.74103585657375, 121.11553784860553, 75.29880478087651 ],
    "bbox_mode": 1,
    "segmentation": [
        [  635.0597609561753,  268.44621513944224,
           636.2549800796812, 194.74103585657375,
           756.1752988047808, 196.7330677290837,
           753.7848605577689, 270.03984063745025
        ]
    ],
    "area": 8776.25
},
```


Figure 11 – Example of a correctly formatted annotation in the COCOjson label file

Firstly, **finalCOCO_dataprocessing.py** contains three helper functions that assist in the data formatting and calculations. First is **write_to_file()**, which is very simple, and just takes in a file path and some content to write to the file at that file path (in this case, a .txt file containing the data formatted in COCOjson form). Next is **extract_segmentation_and_bbox()**, which takes in a .json file path and extracts the segmentation and bbox data from each annotation entry, as well the name of the image file that the current .json file is labelling. Finally, there is the function **polygonarea_and_segmentationformat()**, which takes as an input a string that represents the segmentation points before they have been correctly formatted. This function strips the string down to its individual number values and then reconstructs them into a list that is returned by the function; that list is now in the proper COCOjson format. Once the list was assembled, the area of the polygon was calculated using the Shoelace Formula; both of these values are then returned by the function.

After these three helper functions comes the basic main method, where all of the helper functions are actually called and used. In this main part of the program, the first step is looping through all of the files in the currently specified folder directory (thus helping to automate the process). For each file in the folder, the program checks to see if it is a .json file so that only .json files get read, and then calls the **extract_segmentation_and_bbox()** function to pull the segmentation, bbox, and image file name from the current .json file. Then, the program specifies an output file path for the .txt file this data will be written to. Next, each annotation entry is looped through, and its data is added to an output string, which is then written to the text file just described. The result of all of this processing is .txt files in the **COCO Format** folder

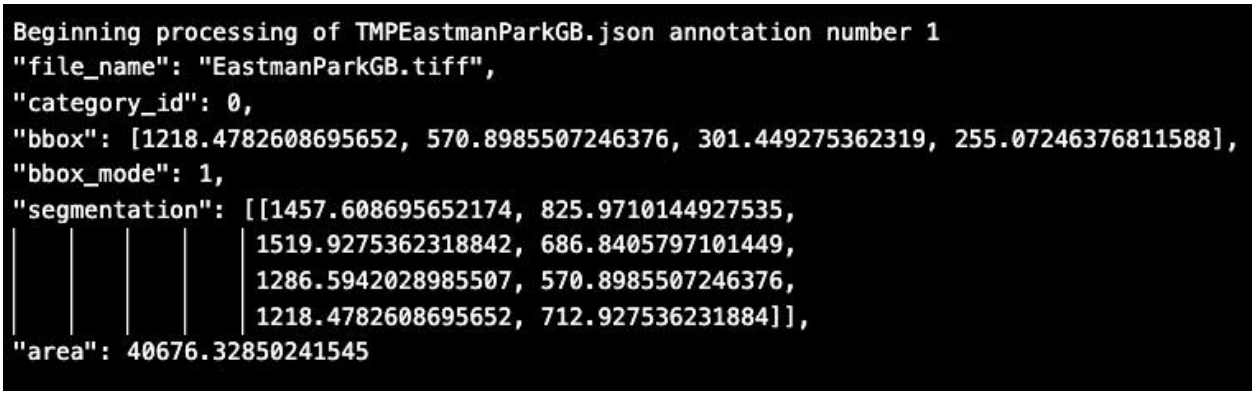


Figure 12 - Contents of .txt file after formatting **segmentation** *and calculating* **area**

that contain label portions that are in the proper COCOjson format (see Fig. 12 for example of an output text file).

After this step, the only thing left to do was copy this data over to the dataset label file, all while ensuring to update the **`id`** and **`image_id`** values for each entry, and those values change for each annotation. While automating the process of adding this data to the .json file was considered, it was determined that doing each label manually would ensure better control and accuracy over the contents of the label file. Fig. 13 shows the resulting label that was added to the .json file that contains all of the labels for the images in the dataset, while Fig 14. shows how the image file is specified in the images portion of the COCOjson file.

After this, the only thing left to do was verify the location of the labels on the original image; that is, the labels need to be visualized over the top of the image they are labelling to


```
{
    "id": 24,
    "image_id": 11,
    "file_name": "EastmanParkGB.tiff",
    "category_id": 0,
    "bbox": [
        1218.4782608695652,
        570.8985507246376,
        301.449275362319,
        255.07246376811588
    ],
    "bbox_mode": 1,
    "segmentation": [
        [
            1457.608695652174,
            825.9710144927535,
            1519.9275362318842,
            686.8405797101449,
            1286.5942028985507,
            570.8985507246376,
            1218.4782608695652,
            712.927536231884
        ]
    ],
    "area": 40676.32850241545
},
```


Figure 13 – Label after being added to the .json file containing all labels for the whole dataset


```
{
    "id": 9,
    "file_name": "Irvine_NoCourt.tiff",
    "height": 2160,
    "width": 3840
},
{
    "id": 10,
    "file_name": "354.jpg",
    "height": 602,
    "width": 939
},
{
    "id": 11,
    "file_name": "EastmanParkGB.tiff",
    "height": 2160,
    "width": 3840
}
```


Figure 14 – Specifying a new image file in the **`images`** *portion so the detectron2 framework knows with which image each label is associated*

```python
# Get file path to json label file
labels_path = "/Users/jacob/Desktop/CPSC 597/detectronDataset/aaaddd.json"

# Read the dataset dictionary from the JSON file
with open(labels_path, "r") as f:
    dataset_dict = json.load(f)

# Prompt the user to enter the image file name
image_file_name = input("Enter the image file name: ")

# Find the image ID for the specified image file name
image_id = next((image["id"] for image in dataset_dict["images"] if image["file_name"] == image_file_name), None)

# Check if the image ID is found
if image_id is not None:
    # Find annotations for the specified image
    annotations = [ann for ann in dataset_dict["annotations"] if ann["image_id"] == image_id]

    # Read the specified image
    img = cv2.imread("/Users/jacob/Desktop/CPSC 597/detectronDataset/" + image_file_name)

    # Visualize the specified image with its associated labels
    visualizer = Visualizer(img[:, :, ::-1], scale=0.5)
    out = visualizer.draw_dataset_dict({"annotations": annotations})
    # Give window a title that includes the current image file name
    window_title = image_file_name + " Plus Label(s)"
    # Show window and keep displayed until user presses a key
    cv2.imshow(window_title, out.get_image()[:, :, ::-1])
    cv2.waitKey(0)
# If the image ID was not found, display an appropriate error message
else:
    print("Image file name not found in annotations.")
```

Figure 15 - Contents of the **`verify_labels.py`** *file*

ensure the labels are in the correct spot. This verification and visualization were done using the helper program **`verify_labels.py`** (Fig. 15).

As this figure shows, the program starts out by specifying a label file to read the labels from, and then asks the user to input an image file name to be loaded. Next, the .json label file is checked for the existence of an image with that same name; if one exists, the value resulting from this is stored in **`image_id`**. After this, there is a conditional statement to check to see if the **`image_id`** is **`None`** or if it has contents. If it is **`None`**, then no images with that same name were found in the label file, and the image loading will be skipped. If **`image_id`** contains data, however, the other part of the conditional statement will first pull the annotations for the specified image, after which the image file with the name specified by the user will be read. Then, a **`Visualizer`** instance is created to store the image file, and the **`out`** variable is created to draw the labels on the image stored in the **`visualizer`** instance. This **`out`** variable is then

drawn using the **`cv2.imshow()`** function. Figs. 16 and 17 show some examples of verifying labels using the **`verify_labels.py`** program.

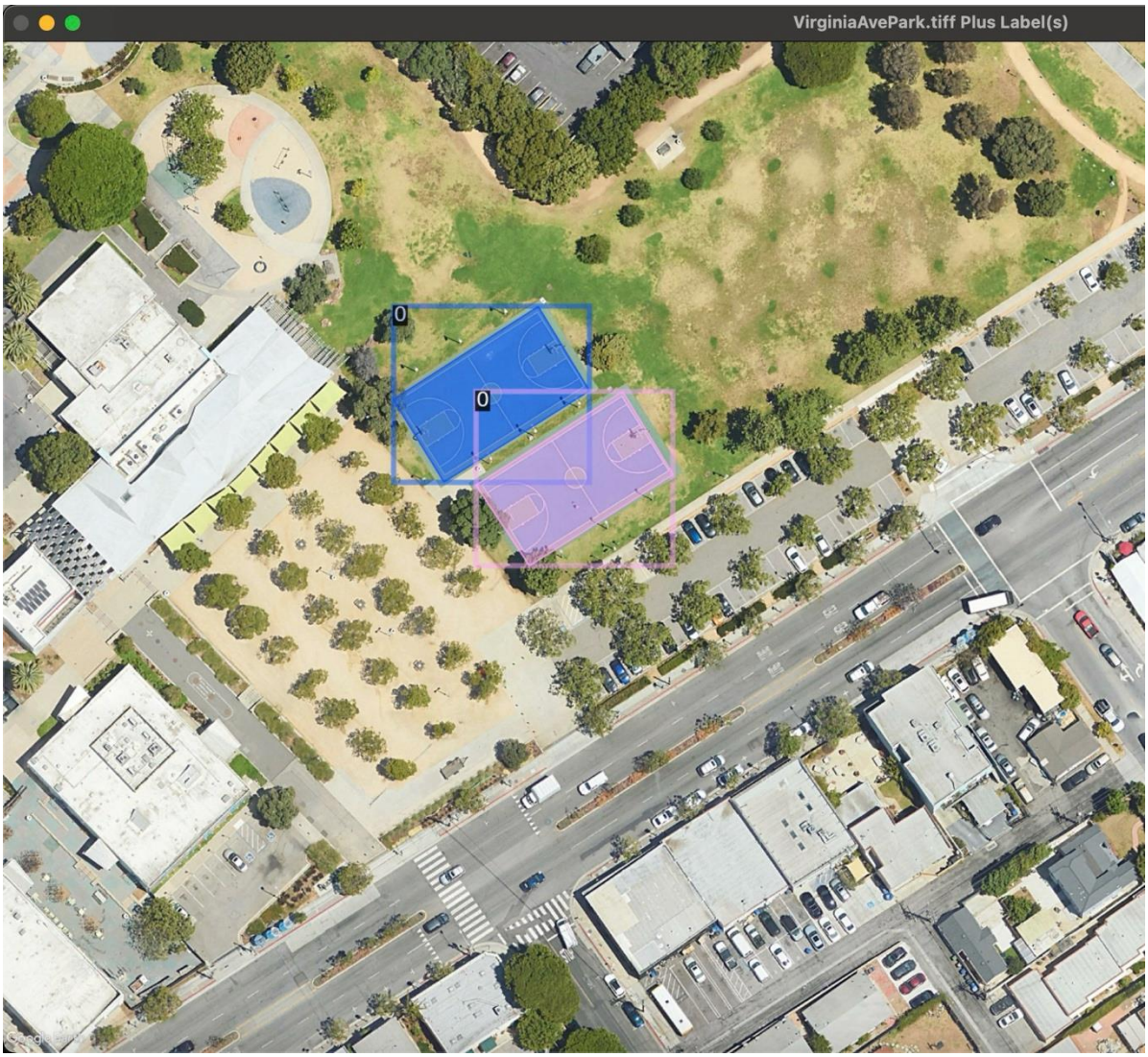


Figure 16 – Verifying labels for the **`VirginiaAvePark.tiff`** *file*

As the figures show, the basketball courts within them are labeled using the bounding box and segmentation points. Fig. 16 provides a good visual example of the distinction between the bounding boxes and polygons discussed in this project. As previously discussed, bounding boxes will always be square with the edges of the image, so this would be the thin outlines in Fig. 16. Meanwhile, the polygon gives a better idea of the actual shape of the object, and in Fig. 16 the polygons are the colored in portions that are more directly on top of the court. In addition, the "0" label associated with the bounding box simply refers to the **`category_id`** that the

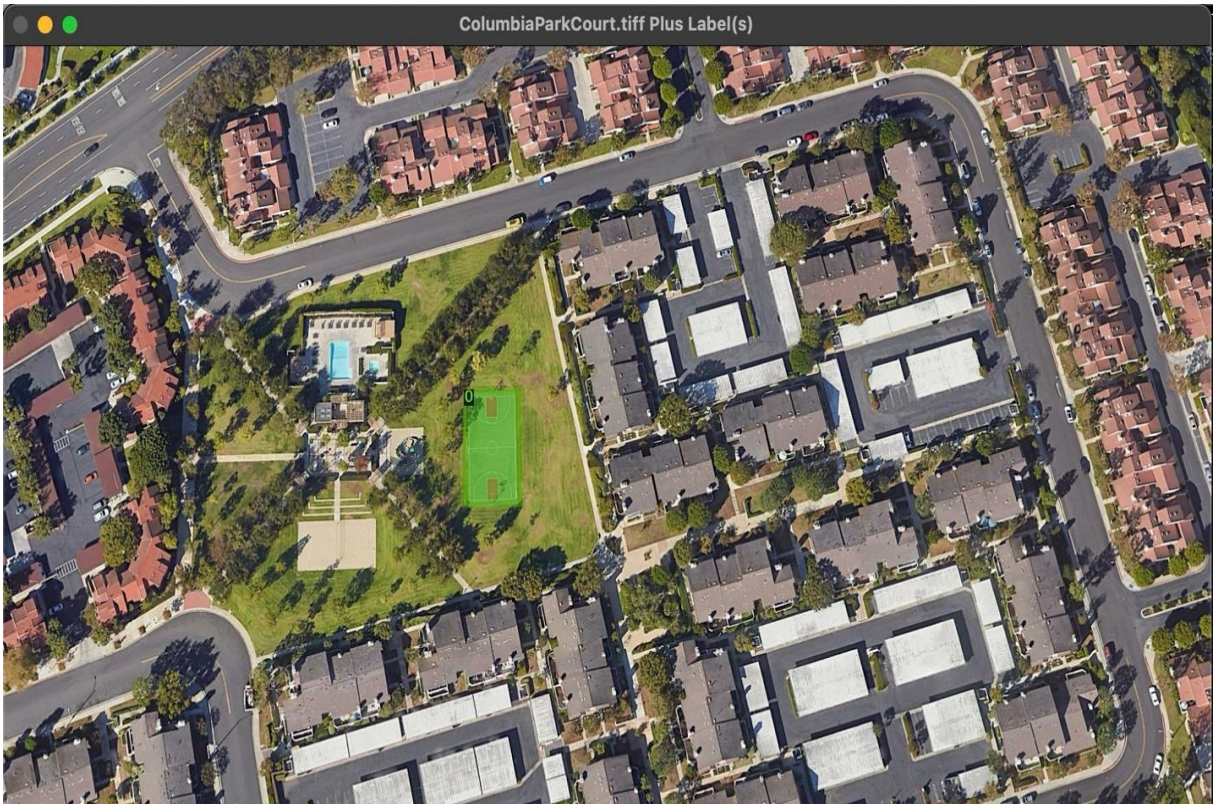


Figure 17 – Verifying labels for the **`ColumbiaParkCourt.tiff`** *file*

object belongs to. In the case of this project, there is only one object being detected, so only one category_id is needed to describe that object. After verifying the labels, the process of labeling and adding an image to the dataset was complete; the process was repeated multiple times with multiple batches of images.

NWPU VHR-10 Dataset

While creating custom images with custom labels was not an overly difficult process, it was nonetheless quite tedious and time consuming, which made the discovery of the NWPU VHR-10 Dataset even more valuable. As previously mentioned, this dataset contains a wide variety of satellite imagery and labels for a wide variety of objects throughout all the images, and one of these object categories was "basketball court." Additionally, the label file provided with the dataset was already in COCOjson format; this made it much easier to add files from the NWPU VHR-10 Dataset to the project's custom dataset. All that needed to be done to add these VHR-10 files to the custom dataset was: pull each annotation with a **`category_id`** matching the **`category_id`** representing basketball courts, determine which image each annotation is referencing, copy said data to the custom dataset, and then copy the correct image to the new

dataset. The first part of this process was done using another helper program called **`NWPUdata_extract.py`** (Fig 18). This program begins by opening the label file from the NWPU VHR-10 dataset, after which it initializes a set that will store the image IDs of images that contain basketball courts. Then, all of the annotations in the file are looped through, and each annotation with a **`category_id == 6`** (which represents basketball courts) has its respective **`image_id`** pulled. Then, all of these **`image_id`**s are used to pull the corresponding image data from the **`images`** portion of the .json file. After this, the annotations are looped

```python
def extract_annotations_with_category(json_file_path, category_id_to_extract, output_file_path):
    # Open the JSON file and load the data
    with open(json_file_path, 'r') as f:
        data = json.load(f)

    # Initialize a list to store the image IDs of the annotations with the specified category_id
    image_ids_to_include = set()

    # Loop through the annotations and extract the image IDs of those with the specified category_id
    for annotation in data["annotations"]:
        if annotation["category_id"] == category_id_to_extract:
            image_ids_to_include.add(annotation["image_id"])

    # Filter the "images" portion to include only those referenced by the annotations with
    # the specified category_id
    images_to_include = [image for image in data["images"] if image["id"] in image_ids_to_include]

    # Initialize a list to store the extracted annotations
    extracted_annotations = []

    # Loop through the annotations and extract those with the specified category_id
    for annotation in data["annotations"]:
        if annotation["category_id"] == category_id_to_extract:
            extracted_annotations.append(annotation)

    # Create a new dictionary containing the extracted annotations and filtered images
    extracted_data = {
        "info": data.get("info", {}),
        "licenses": data.get("licenses", []),
        "images": images_to_include,
        "annotations": extracted_annotations,
        "type": data.get("type", "instances")
    }

    # Write the extracted data to a new JSON file
    with open(output_file_path, 'w') as f:
        json.dump(extracted_data, f)
```

Figure 18 – Contents of the **`NWPUdata_extract.py`** *file*

through again; this time, each annotation with a **`category_id == 6`** is pulled and added to the extracted data.

After this, all that is left to be done is add the pulled image data and annotations to a structure, which is then written to the output .json file; as previously stated, the segmentation and bbox points for these values are already in COCOjson format, and thus did not require any additional processing. After pulling the relevant labels and their corresponding images, the only thing left to do was to add the image files and their corresponding labels to the dataset folder and label file, respectively. While it likely would have been possible to further automate this process, it was determined that adding the labels manually would decrease the likelihood of errors being added to the label file, and would also ensure that the **`image_id`** portions lined up with the correct labels.

The result of this entire process is a folder named detectronDataset, which is a custom dataset containing the test data, its labels, and a folder named Test where the images to be predicted on are stored. The images consist of 99 image files, with 97 files displaying 176 individual basketball courts. These image files are either in the .tiff or .jpg format, with .tiff images being the custom ones created for the dataset, and the .jpg images being the ones added from the NWPU VHR-10 dataset. The test folder contains ten images that the model can make predictions on; seven of these images actually contain basketball courts, while the other three do not. The labels for all of these images (only Training images are labeled) are contained in a single .json file that follows COCOjson formatting rules.

Model Construction

After the dataset that would be used to train the model was created, the next step in the project was to actually train the model on said dataset while making fine-tuning adjustments to the model parameters in order to improve the training and accuracy of the model. As previously discussed, the model for this project is based on a model provided in the “Detectron2 Tutorial”

```
[ ] register_coco_instances("BasketballCourt_Sat", {}, "/content/drive/My Drive/detectronDataset/aaaddd.json", "/content/drive/My Drive/detectronDataset/")
    registered_datasets = DatasetCatalog.list()
    print(registered_datasets)

    ['coco_2014_train', 'coco_2014_val', 'coco_2014_minival', 'coco_2014_valminusminival', 'coco_2017_train', 'coco_2017_val', 'coco_2017_test', 'coco_2017_

    # Get cfg var for setup
    cfg = get_cfg()
    # This imports a pre-trained model config (acts as the model's backbone)
    cfg.merge_from_file(model_zoo.get_config_file("COCO-InstanceSegmentation/mask_rcnn_R_50_FPN_3x.yaml"))
    # Specify the dataset to train on
    cfg.DATASETS.TRAIN = ("BasketballCourt_Sat",)
    # Did not implement custom test dataset, predictions were done in a different file
    cfg.DATASETS.TEST = ()
    # Where to save the trained model
    cfg.OUTPUT_DIR = "/content/drive/My Drive/trained_models"
    # Default value suggested by tutorial
    cfg.DATALOADER.NUM_WORKERS = 2
    # Import pre-trained model weights from same model in model zoo
    cfg.MODEL.WEIGHTS = model_zoo.get_checkpoint_url("COCO-InstanceSegmentation/mask_rcnn_R_50_FPN_3x.yaml")
    # Do not filter out empty annotations to avoid filtering out images that do not contain basetkball courts
    cfg.DATALOADER.FILTER_EMPTY_ANNOTATIONS = False
    # This is the batch size
    cfg.SOLVER.IMS_PER_BATCH = 2
    # This is the learning rate
    cfg.SOLVER.BASE_LR = 0.0003
    # Max number of iterations
    cfg.SOLVER.MAX_ITER = 600
    # Do not decay the learning rate
    cfg.SOLVER.STEPS = []
    # Default is 512, but using smaller values because of the smaller dataset
    cfg.MODEL.ROI_HEADS.BATCH_SIZE_PER_IMAGE = 128
    # Set number of classes in dataset - since we are only detecting basketball courts, this is 1
    cfg.MODEL.ROI_HEADS.NUM_CLASSES = 1
```

Figure 19 – Registering dataset with detectron2 and setting up model parameters/arguments

notebook, specifically the portion that trains the model on the balloon dataset. Fig. 19 provides a look at the setup of the model that was trained for this project.

Before the model can access the dataset specified in **`cfg.DATASETS.TRAIN`**, the dataset in question must be registered with detectron2 (this is the case for all datasets that are not default-included with the framework). The registering of the detectronDataset is done at the top of Fig. 19, and only takes one line, where the user must specify the path for the dataset folder, the labels for that dataset, and then a formal name for the dataset. After this, it was time to set up the model. This began by instantiating a configuration object (**`cfg`**) using **`get_cfg()`**; the parameters of the model are accessed and assigned using the **`cfg`** object. As Fig. 19 shows, the model construction begins by setting up the configuration for the model – in the case of this model, the configuration settings were imported from a pre-trained model from the detectron2 model zoo. After this, a training dataset is defined (in this case, the custom dataset that was just registered with detectron2). Then, the path at which to save the trained model is specified, after which the **`DATALOADER.NUM_WORKERS`** value is set to two (this is the number of cores doing work in training, and two was the suggested value from the tutorial notebook). After this, model weights are imported from the same pre-trained model from the model zoo as the configuration import. The **`FILTER_EMPTY_ANNOTATIONS`** value is next, and this is set to **`False`** so that images that do not have annotations (i.e., images that do not contain basketball courts) are not

filtered out by the model. Next came the Items per Batch (**SOLVER.IMS_PER_BATCH**), learning rate (**SOLVER.BASE_LR**), and the maximum number of iterations (**SOLVER.MAX_ITER**). After this, **SOLVER.STEPS** is set to **[]**, meaning that the learning rate will not be decayed. Finally, the batch size (**MODEL.ROI_HEADS.BATCH_SIZE_PER_IMAGE**) is set. The last thing that is declared for the model is the number of classes (**MODEL.ROI_HEADS.NUM_CLASSES**), and since this project is focused on recognizing one type of object in images, only one class is needed.

Initial testing with this base model revealed an issue that was resolved in Fig. 20. Specifically, the model was giving warnings and errors about the number of output classes not matching the number of classes that the model was able to predict. This almost certainly originated from the fact that a pre-trained model was used to initialize the configuration and weights of the model. That is, the pre-trained model was certainly trained on a larger dataset with more object classes being recognized and predicted, which means that its final layer (and the output dimension) expected many possible output classes instead of just one, meaning its output dimension was too large. As discussed, Fig. 20 describes the modification that was used to fix this error. The second section of code in this screenshot is the portion that replaces the last layer of the model with a new layer that is expecting only one class. The **in_features** attribute of the model represent the number of dimensions (number of classes) that are expected as an input to the classification layer (the final layer of the model). After accessing this value, it, along with the **MODEL.ROI_HEADS.NUM_CLASSES** object from the model creation, are used to create a

```
# Instantiate model
model = DefaultTrainer.build_model(cfg)

# Replace the last layer with new layer for 1 class
in_features = model.roi_heads.box_predictor.cls_score.in_features
model.roi_heads.box_predictor.cls_score = torch.nn.Linear(in_features,
                                                          cfg.MODEL.ROI_HEADS.NUM_CLASSES)

# Freeze the weights of the backbone network
for param in model.backbone.parameters():
    param.requires_grad = False

os.makedirs(cfg.OUTPUT_DIR, exist_ok=True)
trainer = DefaultTrainer(cfg)
trainer.resume_or_load(resume=False)
trainer.train()
```

Figure 20 – Executing command to train model, correct the number of classes in the last layer of model, and freezing weights

new Linear layer that will be the classification layer of the model. This newly created layer is then assigned to the **`cls_score`** layer so it can serve as the final layer of the model.
In addition to replacing the last layer of the model with a new layer predicting the correct number of classes, freezing the weights of the backbone of the network was experimented with. Freezing these layers seemed to result in more accurate predictions that not freezing them, which is why this code was included. After freezing the weights, the model was trained by calling **`trainer.train()`**. Figs. 21 and 22 give an idea of what the training process of the model looked like. The progress of the model's training was tracked using the command line outputs. An attempt to use the tensorboard plugin was made, but issues connecting it to the model were encountered, so this analysis was omitted due to time.

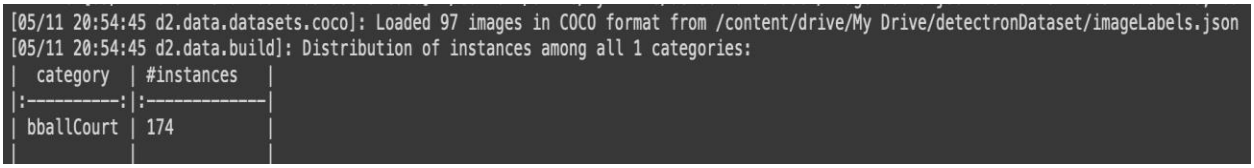


Figure 21 – Number of instances pull from the label file by the training model

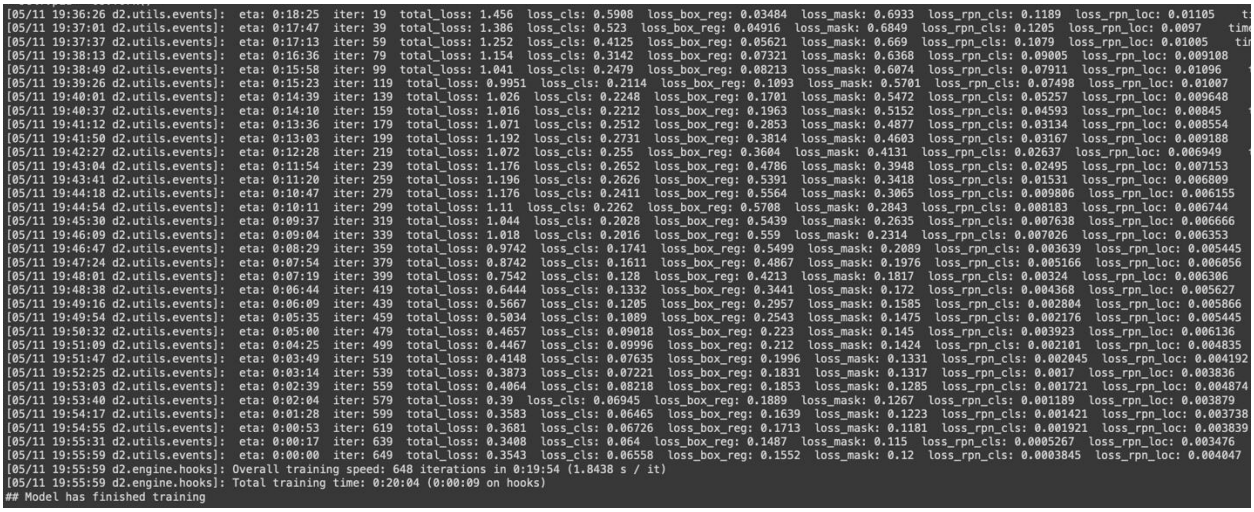


Figure 22 – Tracking loss throughout training process

As can be seen in Fig. 21, the model was able to load 174 of the 176 annotations on the images for use in training. The cause of the two missing annotations was unfortunately never identified. The fact that the model is able to access and load the labels shows that the labels and images are in the correct format for the detectron2 framework. Meanwhile, Fig. 22 shows the progress of the model's training as outputted to the terminal – this view displays many pieces of information, such as the total loss, the iteration number, and other loss data.

Model Testing and Model Predictions

The chosen values of the hyperparameters described above (and shown in Fig. 19) resulted from lengthy experimentation with the hyperparameter values and the pre-trained models used as the backbone for this model, with this combination of values resulting in the lowest overall loss from the model training. As Figs. 21 and 22 show, the final loss for the model was .3453, which is pretty average. However, considering this model was trained on a custom dataset, this final loss value does not look as bad.

As was also previously discussed, a Test dataset for this project was not implemented due to time constraints and the fact that it was quite easy to write a simple program that loaded in the trained model information and then used that model to make predictions on an image file. These predictions are done using the **`detectron2_model_predict.py`** program (Fig. 23).

```
# Get input from user on which image should be predicted on. Also include a message to the
# user to ensure the files are in the proper directories
image_to_predict = input(
    "Please input an image file to predict on.\n Please ensure that the file name is in the \"Test\" folder"
    " of the detectronDataset, and that the dataset folder is in the same directory as this file: ")

# Construct full image filepath:
img_filepath = '/Users/jacob/Desktop/CPSC 597/detectronDataset/Test/' + image_to_predict

# Get cfg to work with model
cfg = get_cfg()
# Predictions only need to use the cpu
cfg.MODEL.DEVICE = "cpu"
# This line is supposed to filter out predictions with a certainty score lower than 40%,
# but it would not work and kept showing no predictions at all
# cfg.MODEL.ROI_HEADS.SCORE_THRESH_TEST = 0.4

# Get the model configuration from the model zoo
cfg.merge_from_file(model_zoo.get_config_file("COCO-InstanceSegmentation/mask_rcnn_R_50_FPN_3x.yaml"))
# Get the trained model weights
cfg.MODEL.WEIGHTS = "/Users/jacob/Desktop/model_final.pth"
# Create a predictor instance that will actually make the prediction
predictor = DefaultPredictor(cfg)
# Read the image file that will be predicted on
im = cv2.imread(img_filepath)
# Use the predictor instance to make a prediction on the image, store the predictions in "outputs"
outputs = predictor(im)

# Create a visualizer instane to visualize the predictions
v = Visualizer(im[:, :, ::-1], scale=0.5)
# Draw the predictions on the image, store the image + predictions to a new variable, "out"
out = v.draw_instance_predictions(outputs["instances"].to("cpu"))
# Convert the image + predictions to a regular color image that cv2 can recognize
img = cv2.cvtColor(out.get_image()[:, :, ::-1], cv2.COLOR_RGBA2RGB)
win_title = "Predictions on Image File " + image_to_predict
# Use cv2.imshow to display the image and its prediction
cv2.imshow(win_title, out.get_image()[:, :, ::-1])
cv2.waitKey(0)
```

Figure 23 – Program that makes predictions on image files and displays those predictions

In this program, the user is prompted to enter a file name, and is also instructed on where the image file they are trying to predict on should be located. Then, the full file path for the image is constructed. Then, a **`cfg`** instance is created in order to work with the model; first the **`MODEL.DEVICE`** is set to **`cpu`**. After this, the model configuration is retrieved from the same pre-trained zoo model as in the training, and the model weights are loaded from the trained model at the specified file path. Then, a **`predictor`** instance is created to actually make the predictions, after which the image is loaded, and the **`predictor`** instance is used to predict on the loaded image. Finally, the image and its predictions are displayed to the screen using the **`cv2.imshow`** function. The result of running this file on a test image is something like what is shown in Fig. 24. Additionally, Figs. 25 and 26 show more examples of the model making predictions on images.

Figure 24 – Using trained model to make predictions on **`BaldwinPark.tiff`** *image file*

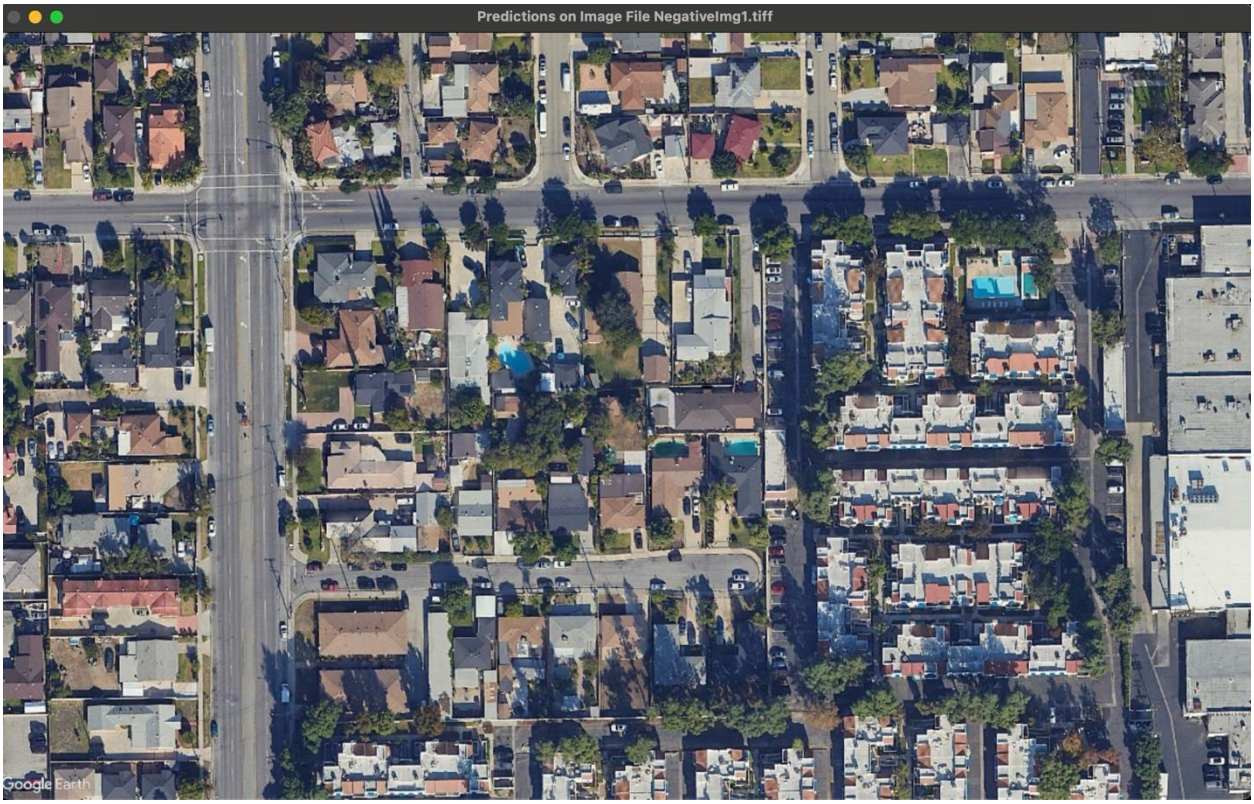


Figure 25 – Using trained model to predict on an image with no basketball court

Figure 26 – Using trained model to predict on the **`RonaldRaegan.tiff`** *image file*

The three above figures give a good picture of what predictions made by the detectron2 model look like: the bounding box surrounds the object and then the polygon inside is supposed to represent the actual object that is being detected. As Fig. 24 shows well, one of the things this model tends to do is make a lot of predictions on the same image. That is, it will overpredict on the same area many times, likely because it has found features that it was looking for. Despite the lower accuracy of this model, one thing that it can still do relatively well is not predict on areas that are not basketball courts. Fig. 25 provides a great example of this, as the image file in that figure does not contain any basketball courts, and as is shown, the model did not predict that anything in the image was a basketball court. The best prediction the model was able to make is represented in Fig. 26 this image clearly shows that the model has found the basketball court in the image and has labeled it appropriately. Despite this, however, the model again shows its tendency to make many predictions in the same area. Even through these described issues, the figures above make it clear that the model is relatively capable of knowing when an image contains no basketball courts at all.

The images predicted on, that is, those shown in Figs. 24 -26, tended to be the images on which the model made its most accurate predictions. The higher accuracy predictions were the reason that these three images were chosen as examples for this project report. The potential reasons for the lower accuracy in predictions on the other images will be discussed further in the next section.

Limitations/Known Issues

One of the largest limitations with this project is, naturally, the small size of the dataset that the model was trained on. Most models with high accuracy are trained on thousands, or even tens of thousands of images to ensure a wholistic picture of the object that is being studied. However, creating a dataset of that size for this project was clearly unreasonable, so a smaller dataset had to be settled for. The small dataset, along with the other limitations discussed below, are almost certainly the cause of the model having a relatively limited accuracy.

Another significant limitation of this project has to do with the lack of data augmentation performed on the images in the dataset. While multiple attempts were made to perform data augmentation on the image files in the dataset using standard **`detectron2.data.transforms`** augmentations, each of these attempts resulted in errors

in processing the images or errors when trying to pass the augmented images into the model. Some examples of attempted data augmentations include **`Resize`**, **`RandomBrightness`**, **`RandomContrast`**, and **`RandomFlip`**. Due to these persistent errors, data augmentation was omitted from this project in order to ensure the model was able to train on the data without errors.

As recently alluded to, the major know issue with this project's model is its relatively low achieved loss and prediction accuracy. The final loss achieved by the trained model was .3453, which isn't all too great; this is almost certainly due to the limitations discussed above. While the model's accuracy may not have been great, Figs. 24-26 still show that the model has made some progress and is somewhat capable of recognizing the features that make up a basketball court.

Another significant limitation of this project is that lack of design artifacts/diagrams. There were plans to construct these figures and insert them into the paper, but there ended up not being enough time to complete these Figures before the deadline due to some extenuating circumstances in other areas.

A final known issue with the project is the lack of an effective user interface that would make the project more interactive and generalizable. Specifically, a user who wants to run the project files locally on their machine will need to go in and edit file paths in certain program files in order to have the program files work for them. Additionally, there is no effective interface that allows users to add their own images and labels to the dataset – this process must be done manually in the same way as was done for this project.

Next Steps/Future Work

There are a significant number of areas in which this project could be improved upon, and there are also many areas that could easily be carried on in future work that could more fully realize the original vision of the project.

The first and most glaring piece of Future Work for this project would be the completion of the design artifacts and diagrams for this project. As previously mentioned, diagrams such as the Use Case Diagram(s) or the Context Diagram were not included in this report due to time constrains from an external extenuating circumstance. If this project were to be continued, the completion of these diagrams would be one of the first things completed.

Outside of this, one of the most significant of these areas has to do with the accuracy of the model, and by extension, the small dataset and lack of data augmentation. As discussed in the Limitations section, the model did not have a great loss rate, .3453, and this was likely due to the small dataset size and the lack of data augmentation. An increase in the dataset size and the implementation of data augmentation would create a more robust dataset with which to train the model, and this would almost certainly produce a more accurate model. Increasing the size of the dataset would not be too difficult, but rather tedious, if the images were to all be selected, labeled, and processed by hand. Thus, it would be ideal if another dataset similar to the NWPU VHR-10 were to be found. However, the problem with almost any other dataset is that their labels will likely not be in COCOjson format, which would require extra time for processing and formatting. However, it is quite certain that more training images would help the model become more accurate and generalized.

As mentioned, another significant area that could be improved upon is the lack of data augmentation. In this case, all that needs to be done is apply the augmentations to the images in the training dataset. This part would not be very difficult, and it certainly would have been quite possible to achieve given more time. The difficult part surrounding the data augmentation would be regarding the expansion of the dataset, as it is somewhat likely that images from different sources (other datasets, not Google Earth Pro) may also initially cause errors with the data augmentation. This could be due to the file type, the image data (color channels, etc.), or any number of other issues. While it would certainly be possible to correct such problems, it may be difficult and time consuming and may also require custom processing for data from each specific source. However, such a process is almost certainly necessary in order to increase the accuracy of the model's predictions.

Outside of the model's accuracy, the other major limitation of this project is a lack of a user interface, which could make it difficult for external users to correctly run or make use of some of the programs included with this project. For example, in the **`verify_labels.py`** file, the image file to be predicted on is provided by the user as an input to the command line. After this, the file path to that image is statically constructed, and as such, another user wishing to run this file will need to change the file path variables in order to route the program to the correct place. In a more robust and generalizable program, it would be ideal for these file paths to be constructed dynamically based on the user's system and the location of the files within their

system. Given more time, a simple window (a File Explorer/Finder window) to browse image files and select them would have been implemented to make some of this project's programs more accessible and generalizable. It would have also been ideal to provide a similar interface that would allow users to specify the label file for images, such as in the **`verify_labels.py`** program (Fig. 15).

Another area for Future Work regarding this project's topic would be continuing working towards developing a front end and an application based off this model, as was discussed in the Initial Project Motivation portion. The most important aspects that would need improvement if a front end were to be fully realized would be: improvement of model accuracy, more generalizable code, and finding a way to provide the model with satellite imagery of the area in question. Increasing the model accuracy would likely not be too difficult and would simply require steps as described above. Additionally, making the helper programs more generalizable would not be very difficult at all, and would just require some re-working of the code. However, finding satellite imagery with high enough resolution would be an extremely difficult task, and would likely be the last, and most difficult, aspect implemented.

Overall, there certainly are some significant limitations when it comes to this project, mainly the model's accuracy and the lack of some diagrams. While it would have been ideal to train a model capable of making higher quality predictions, that was simply out of the scope of what was possible in a single semester, especially considering the time that was spent in constructing the database. However, these limitations (dataset size, lack of augmentation) provide countless areas in which the project could be improved upon or in which future steps can be taken.

Summary and Conclusion

As discussed, this project went through a few different iterations before finally arriving at its final form. The initial goal of this project was to build an app or website similar to www.letsgoball.net or www.courtsoftheworld.com, however upon further consideration, this was determined to be too great of an undertaking for one semester. Thus, this project's goal was modified to be simply building a model capable of recognizing basketball courts in satellite imagery; this model could then in theory be used as the basis for an app or website, as in the original project inspiration.

Initially, TorchGeo was planned as the library that would be used for the project, but due various difficulties encountered, the library was changed from TorchGeo to rastervision, and then from rastervision to detectron2. The change from TorchGeo/rastervision to detectron2 allowed the project to focus on the actual object recognition portion instead of focusing on the geospatial data that is needed for both TorchGeo and rastervision. At the beginning of the project, it was speculated that creating the database on which to train this model would be the most difficult and time-consuming part, and this turned out to be true. Similarly to the library choice, the way in which images were labeled went through several iterations, and eventually landed on the process that is described in the Dataset Construction portion of this report. This process was somewhat tedious, but it does work, and it requires the user to go through manually, meaning errors or other issues are more likely to be discovered. Thankfully, the NWPU VHR-10 dataset was also discovered; this contributed many court images to the custom dataset, making it more robust than it would have been otherwise.

After going through these changes in the project design, the dataset for training the model and the model itself were finalized. In total, the dataset contained 176 courts from 97 image files. This resulted in a model with less than stellar accuracy; the best prediction the model was able to make was on the image file shown in Fig. 26. As mentioned, this is subpar, but is likely due to the small dataset used to train the model, as well as the lack of data augmentation performed on the data. This subpar model leaves open an avenue for future work; that is, this model certainly has room for improvement, and more work (such as adding more images to the dataset or implementing data augmentation) could easily be done in the future in order to increase the model's accuracy.

While this project is certainly far away from becoming something like www.letsgoball.net or www.courtsoftheworld.com, it still offers a different and unique approach to the problem of finding basketball courts in a user's area. As previously described, courts will only appear on these websites if a user has manually added them to the database. However, this model (trained to a better accuracy) could provide a new source of data for website such as this, and this could help in the automation of adding new courts to these online resources. As such, some next steps that could be taken with this project to move it further along would be: creating a larger database and implementing data augmentation to train the model to a better accuracy, making the program files for generalizable so that they can be more easily used by others, and

providing some sort of interface that would allow the model to communicate with websites such as www.letsgoball.net or www.courtsoftheworld.com.

In conclusion, while this project may not have produced a very accurate model, it still proved to be a quality learning experience and provided a new perspective on datasets that are used to train models. Mainly, it gave perspective on how large datasets truly need to be in order to accurately train a model on them. These datasets involve thousands of images, and most have labels, so the sheer scale of a fully sized dataset is much different than the very small dataset used for this project. This project also gave perspective on how some datasets can be created and how time consuming that process can be, even if it is somewhat automated. The construction of the dataset for this project was a significant and time-consuming undertaking that certainly depicts the level of work that went into this project. In addition to giving perspective on fully sized datasets, this project also provided a learning experience regarding using a python library for object detection in images. While there were some lessons learned from the TorchGeo and rastervision library attempts, the most significant learning came from exploring the detectron2 framework and its expected data format.

References

[1] “facebookresearch/detectron2,” GitHub, May 26, 2020. https://github.com/facebookresearch/detectron2

[2] “LabelMe. The Open annotation tool,” *labelme.csail.mit.edu*. http://labelme.csail.mit.edu/Release3.0/

[3] “Use Custom Datasets — detectron2 0.5 documentation,” *detectron2.readthedocs.io*. https://detectron2.readthedocs.io/en/latest/tutorials/datasets.html

[4] “Google Colaboratory – Detectron2 Tutorial” *colab.research.google.com*. https://colab.research.google.com/drive/16jcaJoc6bCFAQ96jDe2HwtXj7BMD_-m5

[5] A. J. Stewart, C. Robinson, I. A. Corley, A. Ortiz, J. M. Lavista Ferres, and A. Banerjee, “TorchGeo: Deep Learning With Geospatial Data,” *GitHub*, Nov. 01, 2022. https://github.com/microsoft/torchgeo

[6] Azavea/Element 84, Robert Cheetham, “Raster Vision: An open source library and framework for deep learning on satellite and aerial imagery (2017-2023).,” *GitHub*, May 07, 2024. https://github.com/azavea/raster-vision (accessed May 11, 2024).

[7] “RFC 7946 - The GeoJSON Format,” *datatracker.ietf.org*. https://datatracker.ietf.org/doc/html/rfc7946

[8] “The Ultimate Guide to COCOJSON: Enhance Your Annotation Management,” *www.deepblock.net*, Dec. 19, 2023. https://www.deepblock.net/blog/the-ultimate-guide-to-cocojson-enhance-your-annotation-management (accessed May 11, 2024).

[9] “COCO - Common Objects in Context,” *cocodataset.org*. https://cocodataset.org/#format-data

[10] “KML to GeoJSON Converter,” *Free File Format Apps*. https://products.aspose.app/gis/conversion/kml-to-geojson (accessed May 11, 2024).

Note: These two citations below are not referenced in text and are simply the citations for using the detectron2 dataset.

[11] Su H, Wei S, Yan M, et al. Object Detection and Instance Segmentation in Remote Sensing Imagery Based on Precise Mask R-CNN[C]. IGARSS 2019-2019 IEEE International Geoscience and Remote Sensing Symposium. IEEE, 2019: 1454-1457.

[12] Su, H.; Wei, S.; Liu, S.; Liang, J.; Wang, C.; Shi, J.; Zhang, X. HQ-ISNet: High-Quality Instance Segmentation for Remote Sensing Imagery. Remote Sens. 2020, 12, 989.