Alex Fish’s Work on Sorghum Panicle Detection

Idea is to use previously published, established object detection algorithms.

Some requirements: Data in an easily computable form, known object locations for training and testing.

The Data

The data comes in the form of 395 images of a 1-acre crop field, taken from above by a drone with a camera pointed straight down. The drone scanned the field across in a snake-like fashion. The images are large, 5472x3648 pixels, and can vary from 6 to 10 MB in size. Not all images have sorghum panicles in them. Some images have up to 10 or 15 panicles. The first image name is DJI\_0001.JPEG and the last is DJI\_0395.JPEG. All subsequent data used comes from these pictures, so they are all that is required to spin-up this project and work on it on your own machine.

Computing on JPEG/PNG images (reading in pixel values, etc.) is easy and therefore not an issue.

The Object Location Data

Because of the large number of pictures and variable number of panicles per picture, we used a tool called Zooniverse which allows many users to help in data collection from images and got help from other student researchers on related projects through James Schnable. Zooniverse requires each image to be 1 MB or less, so the images had to be shrunk down in some way to upload them to Zooniverse. The script **slicepics.R** slices each image into a 5x5 grid and saves each image to a new directory. The first image (top-left corner of the 5x5 grid) is named DJI\_XXXX\_1.JPEG, and the last image (bottom-right corner of the 5x5 grid) is named DJI\_XXXX\_25.JPEG, where the XXXX is the original image’s id.

The resulting images are roughly 500-600 MB and are 1094x729 pixels. The script chooses a select set of images, leaving out images which did not tend to have panicles in them. The range of images chosen was about DJI\_0080 through DJI\_0199 after some quick manual checking. This was done to avoid bloat of the dataset because roughly half of the images do not have panicles or are near identical to previous pictures.

Zooniverse has two options for uploading data, drag-and-drop into their website and their command line tool. The drag-and-drop tool was having issues with the size of the dataset so the command line tool was used. The command line tool requires a manifest, which assigns an id of the user’s choice to each image the user wants to upload. The script **generatemanifest.R** creates this manifest, simply assigning ids starting at 1 and incrementing from there and outputting a file named **manifest.csv**.

The Zooniverse users were asked to click on the centers of any sorghum panicles that were in each image, as well as up to 3 clicks on sorghum plant leaves, and non-sorghum places such as dirt or bushes. Once all the images were analyzed by the users, we requested the data from Zooniverse and downloaded the data. Zooniverse provides the data in a JSON format. The script **readclassificationdata.R** converts the ugly hard-to-use **generate-panicles-training-data-classsifications.csv** into an easier-to-use CSV format, with the following columns: filename, task id, x click location, and y click location. The resultant CSV is named **sorghum\_annotations.csv**.

The script **generatetrainingdataCSV.R** makes a new CSV from the base-data CSV. Bounding boxes of 60x60, centered at each click, are assumed to be accurate for most or all sorghum panicles after manual checking. The resultant CSV is named **flower\_training\_data.csv** with columns filename, top row of bounding box, left column of bounding box, bottom row of bounding box, right column of bounding box, and class. For our purposes, using strictly the sorghum panicle clicks and not the leaves or dirt is sufficient, as sorghum panicles are treated as a “1” class and everything else is treated as a “0” class. As such, all rows have class “flower.”

Attempts at Using Implementations of Object Detection Algorithms

We did some research into a very well-known algorithm, RCNN or Region-Proposal Convolutional Neural Network. There were improvements made to this algorithm over time in the forms of Fast-RCNN, then Faster-RCNN. What takes RCNN 50 seconds to process, Fast-RCNN can do in 15 seconds, and Faster-RCNN can do in 0.2 seconds based on improvements made to the networks’ architectures. Source: <https://towardsdatascience.com/r-cnn-fast-r-cnn-faster-r-cnn-yolo-object-detection-algorithms-36d53571365e>.

Most or all implementations said that Faster-RCNN was deprecated and there were faster, more accurate object detectors out there. The first I saw (<https://github.com/yhenon/keras-frcnn>) pointed me towards RetinaNet (<https://github.com/fizyr/keras-retinanet>). This implementation of RetinaNet is easy to use and install and uses data in the form that **flower\_training\_data.csv** is already in! It also required a “class mapping” file, so it knows what id goes to which class. We manually created **class\_mapping.csv** with one line “flower,1”. However, this network requires up to 11 GB of graphics card RAM, which I did not have easy access to.

I began the search for other algorithms/implementations to use.

Detectron, by FaceBook, was referenced often. Its data input comes in the form of the COCO dataset format. This format is very complex. I found an online tutorial explaining how to create your own (source: <https://patrickwasp.com/create-your-own-coco-style-dataset/>). The tool requires “masks” to be made. A mask is an all-black image of the size of what will be used as data-input to the object detector algorithm, except a white bounding box around a single example. A mask image must be generated for each individual example (each instance of a sorghum panicle).

The script **generatetrainingdata.R** reads in **sorghum\_annotations.csv** and generates the masks for each example and saves them to a specific directory (see the tutorial link for directory structure). The script **generate-coco.py** creates the COCO-style JSON file **instances\_sorghum\_flower\_train2018.json**.

I did not try to use this with Detectron to see if it worked.

Detectron is nice because you can choose from several network architecture backbones to use, including RetinaNet, Faster-RCNN, and an interesting one named MobileNet (made by Google) as well as other high-computational-power detectors such as VGG-16 and YOLO. MobileNet sacrifices some accuracy for the ability to be run on laptops, requiring very little graphics card RAM (around 2-3 GB).

Looking around for implementations of object detectors lead me to Tensorflow’s own implementations of object detectors (source: <https://github.com/tensorflow/models/tree/master/research/object_detection>). They provide their own easy-to-use data converters, to put data in the format they expect as input very easily. Like Detectron, they have many different architectures to use including RetinaNet, Faster-RCNN, VGG-16, YOLO, and MobileNet.

These implementations require a Linux or Mac computer. I did not have time to try to use these.

Notes

All scripts read from and write to folders defined within the scripts themselves. Change the paths within the scripts for your own needs.