Introduction to Google TensorFlow Object Detection Library

Author: Ankan Bhattacharya

Professor: Xiaoyi Lu

Abstract

Object Detection is one of the biggest fields in artificial intelligence and machine learning. It provides considerable real life applications from self-driving cars, medical imaging, and much more. One of the most popular frameworks when it comes to this field is Google's TensorFlow. This is due to the fact that it is highly accessible from running from powerful computers using the top of the line GPUs to even your mobile devices. Despite its flexibility, it does suffer from large performance issues. The following report will explain how to evaluate model performance and look to see its positives and negatives. The model was trained two ways. One on an Intel Core i7 8500 and using the NVIDIA GeForce RTX 3050 Ti. Training on the GPU was around 3x faster than training on the CPU, as the GPU batches instructions and pushes data at higher volumes compared to the CPU. While observing this model, I have come up with three ways to improve the performance. One is to increase the training set size, another is to create a variety of environments for the images, and the final is to increase the number of steps/epochs.

Process - Setup

All the steps that I have taken were done using Nicholas Renotte's TensorFlow Object Detection Course with modifications. There are two notebooks that I followed. The first one is Image Collection and the second one is Training and Detection. Before we start, we need to clone the GitHub repository that the instructor provided. Then we create a virtual environment so we can install all our dependencies in one place while not interfering with other environments. Once we have done that, we install the dependencies and add the virtual environment that we created to the Python Kernel so we can run Jupyter Notebook. After that, we can begin with the first notebook, Image Collection.

Process - Image Collection

These steps are all done using the Jupyter Notebook that Nicholas provided, with minor alterations. The first thing we do is to install opencv-python and import our dependencies. The dependencies we use are opencv, uuid, os, and time. Opencv is used to capture the images, uuid is a universally unique identifier that names our images to have unique names so it is easy for the notebook to differentiate between the different images. The os library helps us perform miscellaneous operating system tasks automatically, and time is used to give us time to set up and take our images. After we import our dependencies, we define the images to collect. I decided to collect images based on the amount of fingers that I'm holding up. **I decided to test 4 different gestures (holding my finger up from 1 to 4) and have 5 different images for each.** After we define the images, we have to set up the folders. The notebook created and implemented an image path that we will use to store the images. After that, we captured the images using opencv2. The way it works is that it will begin by collecting images for a specific number. It then gives you five seconds to set up and take the image, and then it moves on to the next image. It takes the number of images that I specified, which was 5. After capturing the images, we begin to label the images. We had to install pyqt5 and lxml since those are needed to install labelimg, an application we will use to label our images. We create an image path for it and clone the github for labelimg and then open it up.

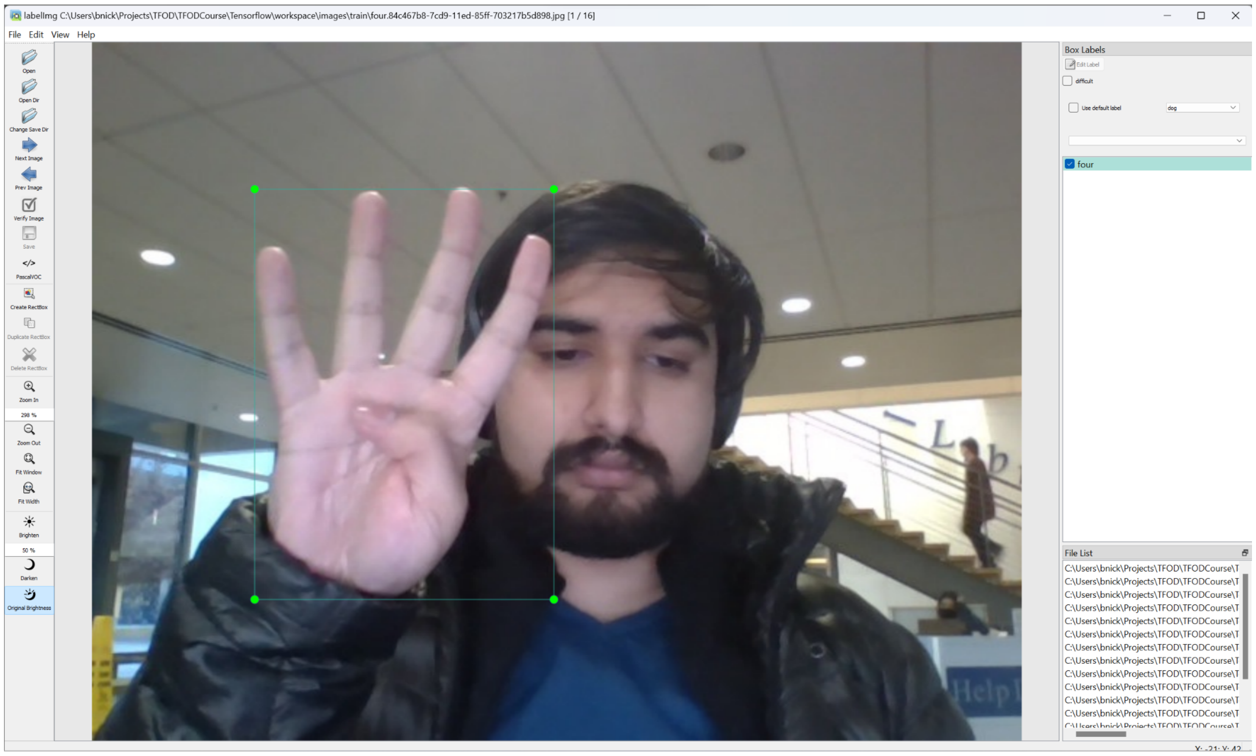
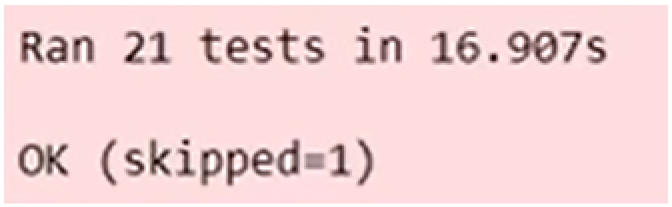


Figure 1: Using labelimg to classify the images.

For each of the images that I took, I had to draw a box around the amount of fingers that I am holding up and label them. **The labels have to be as tight as possible for maximum accuracy.** After that is done, I created a training and testing partition and moved 4 of the images for each amount of fingers I hold for training and 1 of the images for each amount of fingers I hold for testing. After this, I moved on to training and detection.

Process - Training and Detection

Before training and testing, we have to set up our model and paths. **We use the SSD MobileNet V2 model for this, but there are a wide variety of models we can choose.** There are a lot of paths that we have to create for this step, but the Jupyter Notebook provides this. After we set our path names, we create those paths using Jupyter Notebook. After we set our paths we can finally start training and detection. What we first need to do is download the model that we are using from Tensorflow model Zoo and install TensorFlow Model Detection. We first install wget to retrieve files using internet protocols. After that we clone the Tensorflow model repository from GitHub and after that we can finally start installing TensorFlow Object Detection. The process of installing this takes a long time and sometimes skips over installations. The notebook does provide a verification script that does verify if your installation is successful. When I encountered a missing module, I looked up that error and missing module and found the right installation for it. The Jupyter Notebook provides the installations that I have done, but the amount of installations that you have to do is different for everyone. Everything should be installed when at the end of the output of the verification script cell, it says OK.



```
Ran 21 tests in 16.907s

OK (skipped=1)
```

Figure 2. Verification Script confirms that TensorFlow Object Detection is installed

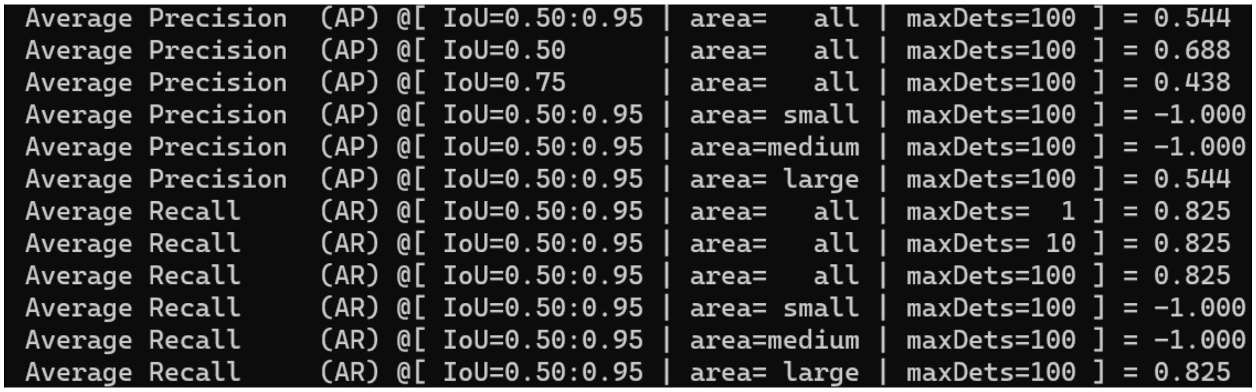
I moved on to download the SSD MobileNet V2 model that I defined as a pretrained model path. After we download everything, we create a label map. A label map represents a map of the labels that I provided. After that we create a TFRecord, which is a binary file format for storing data. We use this to speed up training for the model we are using. It is important that our labels have the same case sensitivity as our images that we created in LabelImg, since it takes the image path of the labels that we used in LabelImg. If it isn't then we will run into an error. After that we copy our model configuration to our training folder and update the configuration for transfer learning. This will set our training model to have the same parameters as the model we are using. After that we can finally train the model. Jupyter Notebook will create a training script and command to utilize, but it is highly recommended to create a new command prompt window and run the command there so we can see what is going on. The command will run our model training script and give it arguments that pass our model directory and our pipeline config and the number of steps we are doing. In this case I did 2000 steps then 10000 steps. The training process is automatic after putting in the inputs. After training, I evaluated the model to get my Average Precision and Recall. I opened this up using TensorBoard as well which provides graphing models.

Process - GPU Implementation and Training

While this step was unnecessary, I did want to work with my NVIDIA GPU to see if it would speed up the process of training and evaluating the model. Depending on the TensorFlow version, there are different versions of CUDA and cuDNN that are needed. I installed tensorflow_gpu-2.10.0 so I needed to install CUDA 11.2 and cuDNN 8.1. After installing that, I noticed my training time improving 3x faster than using the CPU only.

Results

The three main performance metrics that we will focus on are precision, recall, and loss. Our Average Precision is 54.5% and our Average Recall is 82.5% while training for 10000 steps, which can be seen in figure 3.



```
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.544
 Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=100 ] = 0.688
 Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=100 ] = 0.438
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = -1.000
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = -1.000
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.544
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] = 0.825
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] = 0.825
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.825
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = -1.000
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = -1.000
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.825
```

Figure 3. The values of average precision and average recall evaluation of model after 10000 epochs.

Looking at figure 4 and 5, it compares the Average Precision and Recall percentages from 2000 epochs to 10000 epochs. In this example, there is an increase from 39% using 2000 epochs to 54.4% in average precision and an increase from 61% to 84% in average recall. Looking at the loss plot in figure 5, there seems to be a slight decrease in loss.

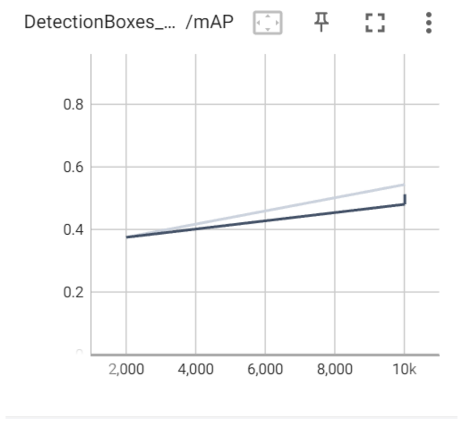


Figure 4. Average Precision Graph from 2000 to 10000 epochs

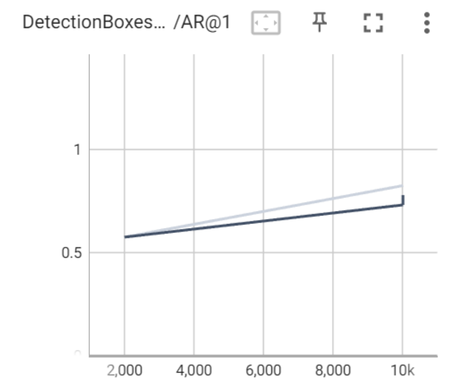


Figure 4. Average Recall Graph from 2000 to 10000 epochs

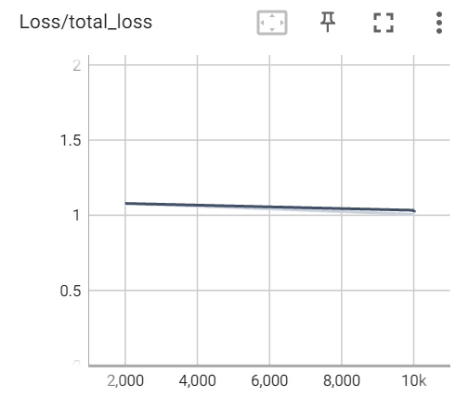


Figure 5. Total Loss from 2000 to 10000 epochs

Observations

The performance metrics help us in improving our object detection model. There are many things we can do to improve the models. One way is to provide more epochs to make the final value of loss further, however there is a rate in which the max number of epochs is reached. Another way is providing more training images with different environments so that there is more data to train on. The video recommended 10-20 pictures with different angles and lighting conditions, but only 5 to run along quickly. Another way to improve the change is the bounding boxes when labeling the images. By using my whole hand, my labels were not as tight as possible. If I just showcased my fingers instead of my hand, it would be a lot more tighter.

Conclusion

This report explains an introductory object detection training and evaluation process. The process gives an introduction to neural networks and deep learning networks and allows us to evaluate the performance of the model as well as providing solutions to improve it. Some ways to improve the model is giving more training images to give it more training time. It will increase the accuracy but the time as well. Another way to improve the model is tightening the bounds of the labelimg to provide more accurate results that the model can test on. Lastly it is recommended to use a GPU to train your model as it is 3x faster than using a CPU.

References

1. "Tensorflow Object Detection in 5 hours with Python | Full Course with 3 projects." YouTube, uploaded by Nicholas Renotte, 9 Apr. 2021, https://www.youtube.com/watch?v=yqkISICHH-U
2. Nicknochnack. (n.d.). Nicknochnack/TFODCOURSE. GitHub. Retrieved December 7, 2022 from https://github.com/nicknochnack/TFODCourse
3. NickB59. (n.d.).NickB59/TFODCOURSE. GitHub. Retrieved December 16, 2022 from https://github.com/NickB59/TFODCourse