Computer Vision Final Project Proposal

Kevin DuCharme Max Li

April 4, 2018

1 Motivation

Robots are quickly becoming a staple in everyones living room, with the likes of Alexa and Google Home acting as virtual assistants that aid in everyday tasks. These first-wave household assistant robots primarily focus on verbal interactions, and are fixed to one location. Recent developments have started to branch out, with computer vision bringing a whole new domain of possibilities to these devices. Jibo, a robot out of the MIT Media Lab, is an example of that. Jibo is a natural extension of the Alexa concept, acting as a mobile home assistant. Computer vision is not only used to navigate, but also for the recognition of common faces in the household, among other functions. While the goal of this project is not completely in line with Jibo, it represents one of the many ways that these personal assistants can intersect with computer vision.

The main motivation behind this project is the struggle of finding items on a heavily used desk. On an active workbench, tools and other useful items are frequently getting picked up, used, and placed down somewhere else. While good standard operating procedures will make sure all the parts end up where they're expected to be, it's very easy to quickly lose track of items.

The longer-term goal of this project was to start the foundation of a lab assistant robot that has the capability of tracking common items found on a workbench in pseudo real-time. When a user queries the robot, it should be able to tell the user where a specific tool is, in a way that would make sense to the user. Initially, the objects being identified would come from a preset bank of objects that have been pre-trained. However, the end product will likely have to be trainable on new objects by the user.

For this project, the main topic of focus was object detection and tracking. After all, the rest of the features are both fairly difficult (and thus time-consuming), and out of scope of the computer vision task. As such, the goal of this project was to be able to detect different objects, identify what they are, and track them as they move across a camera. In order to be consistent with the lab assistant use case, three classes of objects were trained upon: screwdrivers, pliers, and wrenches.

2 Technical Approach

Before getting into the details about the approaches taken for detection and tracking, a list of programming languages and libraries is included to allow for the code to be run.

- Python 3.5
- Tensorflow 1.4.0 (With GPU functionality)
- OpenCV 3.3.10, including extra packages.
- numpy

2.1 Testing Environment

In order to reduce the number of non-idealities to deal with, the test setup was kept as simple as possible. A top-down viewing camera positioned over a desk was utilized to track the trained objects. The camera was a Logitech C920 webcam held in place with a gorillapod tripod. The area the camera covers will be considered the whole workspace and will be well-lit.

2.2 Object Detection

2.2.1 Architecture Selection

The architecture selection was based on criteria for future improvements. While the program was going to be running on a desktop computer for this project, the intended target for this application is a resource-constrained device like a Raspberry Pi. This meant that the solution needed to be lightweight and efficient. Architectures like RCNN and Faster RCNN have proven to show great success with detecting objects, but they can be computationally heavy.

Google released MobileNets, an open-source framework for computer vision models built specially within TensorFlow. These 'mobile-first' models would pair perfectly with resource-constrained devices and are easy to train and deploy. Instead of having a 3x3 convolutional layer followed by a batch norm and ReLU, MobileNets employs separable layers to reduce parameters. This helps make the architecture resource friendly. To continue the trend of lightweight and efficient choices, a single shot multibox detector was used as well. While MobileNets acts as the feature detector, the SSD detects the objects.

2.2.2 Retraining Model

Because of limitations in time and resources, the project retrained a model that was initially trained on the Common Objects in Context (COCO) dataset. In order to do so, a new dataset first needed to be compiled. With wrenches, screwdrivers, and pliers as the target objects, images of these objects were collected. Using Google and Bing image searches as well as Image-net.org, around 150 images were collected of each category. To avoid bias in the model, it is good practice to have an equal number of examples for each category. Additionally, having more examples helps build a more generalized model, though anywhere from 100 to 500 examples will work.

After collecting the new dataset, each image needed labels and boundary boxes for training. LabelImg, by github.com/tzutalin, was used to provide these boundary boxes and labels. An XML file is created for each image and all XML files where then converted to training and testing tensorFlow record files. These record files are the means of importing

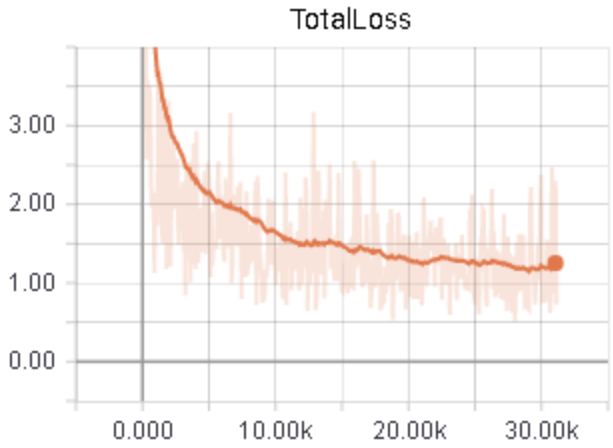

Figure 1: Training Model - Total Loss

the data into tensorFlow for training. Additionally required is a pbtxt and configuration file. The pbtxt file provides a key to connect the neural network output to a classification label, while the configuration file details the architecture of the network. Once these files where collected, the model was then trained for 13 hours and was ready for deployment. The total loss during training the model can be see in Figure 1.

2.3 Object Tracking

The main reasons to use an object tracking algorithm were twofold: to increase the speed of object tracking, and to have identity persistence for individual tracked objects. An increased speed allows for implementation in embedded, resource-constrained systems, while a persistent identity for individual objects creates the framework for locating specific misplaced items.

In order to get a wide variety of object tracking algorithms to choose from, OpenCV 3.3.10 was used. Its ubiquity and Python library made it a reasonably easy decision. However, The OpenCV tracker algorithms take the bounding box locations in the form of pixel values. Tensorflow, however, uses a normalized value between 0 and 1, where (0,0) is the top left corner, and (1,1) is the bottom right corner. Because of this, a quick and dirty conversion function was written, which converts between the two formats.

2.3.1 Tracking Algorithm

With OpenCV 3.3.10, there are 6 different tracking algorithms: Boosting, Multiple Instance Learning (MIL); Kernelized Correlation Filters (KCF); Medianflow; Training, Learning, Detection (TLD); and Goturn. Except Goturn, all of these are trained online. The first three are based on AdaBoost, an ensemble learning method that creates a large number of weak classifiers, and forms a linear combination of them to produce the strong classifier used in tracking. Boosting uses these classifiers in a fairly straightforward manner, trained on

Haar-like wavelets, HoG, and Local Binary Pattern features. The other methods use smart ways of training selection to improve upon performance, with KCF being the fastest and most accurate of the three. Of the three remaining, Medianflow was the most interesting, in part because it was capable of error reporting, but also due to simplicity of implementation. In the end, this is the algorithm selected.

2.4 Detected Object Structure

Within Tensorflow, there is an Object Detection API, which provides an example program. This served as the base from which we expanded upon.

Each detected object is kept in a python class, initialized with its bounding box location (normalized to 1), its class, the score the detector gave to the box, and the tracker type to use. An ID number is also assigned. All the objects are kept in a list.

After initialization, the program is split into two main sections: object detection and object tracking. Object detection occurs once every 30 frames. It is used to update the trackers, which accumulate errors over time. For each object that passes a certain threshold, it is compared to the current list of existing objects. If the bounding box is near an existing one, the existing object is updated. Otherwise, a new detected object is added. At the end, objects that weren't detected are removed. Object tracking occurs for the remaining 29 frames. This goes through the list of objects and updates each tracker.

3 Key Results

Using Medianflow, the average FPS of the detection/tracking workflow was around 110 FPS. With stationary objects the tracking worked as well as the detector, and IDs were maintained. However, with fast motion or occlusion, the ID was no longer maintained. This is likely to do with the tracker not being the ideal one for those issues, as well as the likely not being trained to optimality, dropping the detection of some objects.

4 Distribution of Responsibilities

Kevin collected and labeled the data that was used to conduct transfer learning. He also trained the model with the data, and created the foundation of the program that implemented object detection for the frames from a streaming webcam.

Max incorporated the object tracker into the design, as well as investigating which object tracking algorithm would be ideal. He also created the structure that enabled the ID persistence, writing the helper functions, the DetectedObject class, and the framework that switches between detection and tracking.

References

[1] A. G. Howard et al, "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications," 2017.

[2] W. Liu et al, "SSD: Single Shot MultiBox Detector," 2015.

[3] Helmut Grabner, Michael Grabner, and Horst Bischof, "Real Time Tracking via On-Line Boosting".

[4] Boris Babenko, Ming-Hsuan Yang, and Serge Belongie, "Robust Object Tracking with Online Multiple Instance Learning," 2010.

[5] Joao F. Henriques et al, "High-Speed Tracking with Kernelized Correlation Filters," 2014.

[6] Bryan Anenberg and Michela Meister, "Tracking-Learning-Detection," 2011.

[7] David Held, Sebastian Thrun, and Silvio Savarese,"Learning to Track at 100 FPS with Deep Regression Networks," 2016.