Digital Orientation Guide: Final Report

Section 1: Development Team

Selin Onal

- Speech UI
- Text to command processing
- Visual UI
- General Android upkeep

Rick Sear

- Tensorflow data pipeline (code that goes from dataset to .tflite model)
- Custom Tensorflow object detection for street signs
- Priority queue/priority map for camera regions
- Voice commands hub function/string parsing

Jack Sloane

- Tensorflow object detection for common objects
- ARCore collision avoidance module
- Integration of app modules (camera, collision, help, home) using Android Fragments
- General Android housekeeping/bug fixes

Section 2: Non-Technical Overview

Project D.O.G. (Digital Orientation Guide) is an Android application that uses AI and Image Analysis to simulate the functions of a seeing eye dog through a phone's camera. By analyzing the camera feed of the user's outward facing camera, D.O.G. 's object detection is able to identify most common objects and street signs, as well as the relative position of these objects to the user. D.O.G. is also capable of checking for and alerting of potential collisions and dangers through its collision detection module that measures the distance between objects in the environment and the user.

Project D.O.G. is fully accessible, through both the base design of our app, and through its integration with Android's base accessibility settings, namely, Google Talkback. The app's UI is extremely simple, and all instructional text can be interacted with by touch to remind the user of all the different actions available to them. The app is also notably completely navigable via voice commands, so a user also has the option of avoiding UI interaction altogether. Our app is compatible with all versions of Android starting from Android Oreo (8.0) and up - meaning that it supports the majority of Android devices dating as far back as 2017. Our hope is for D.O.G. to be an accessible, safe, and effective guide for users with visual impairments in any use case.

Section 3: Project Website

Link to website: https://searri.github.io/project-dog/

This website includes a front page featuring our 5 and 10-minute videos, an About page with team members' bios, and a Documentation page with links to PDFs of our writing assignments.

Section 4: Development Tools

- To develop our app we used Android Studio, which includes the Android SDK
- We used the Tensorflow Lite API to perform all object detection in our project
  - TFLite requires a trained object detector model, which we trained with the Tensorflow Object Detection API. Most of the scripts included in our repository can also be found in Tensorflow's official repository but we included them for convenience
- For common object detection, we used a pre-trained single shot detector (SSD) object detection model trained on the Coco dataset using the Mobilenet model architecture.
  - The model file can be found here
  - The Coco dataset can be found here
- For street sign object detection, we trained our own SSD model
  - We warm-started training using checkpoints from "SSD MobileNet V2 FPNLite," which can be downloaded from the Tensorflow Model Zoo
  - We used the Mapillary Traffic Sign dataset to train it
- We used the ARCore Depth API for the collision avoidance module
- We used Google SpeechRecognizer API for text-to-speech and speech-to-text

Section 5: Technical Overview

DOG is composed of four modules: object detection; collision avoidance; home, and help. The app uses one main activity which is the parent of four interchangeable fragments. These fragments host the four modules, respectively. The main activity controls the carousel wheel of buttons used to switch which fragment is active and displayed above the carousel wheel - only one fragment is active at a time. In addition, voice commands can be used to change the active fragment. Voice-to-text and text-to-speech are implemented using the Google SpeechRecognizer API, which streams audio to Google servers to perform speech recognition. As such this API is not intended to be used for continuous recognition, which would consume a significant amount of battery and bandwidth. Instead, voice recognition is temporarily activated by pressing the voice command button on the carousel wheel.

The object detection module uses the Tensorflow Lite machine learning API. Using the camera's live video feed, thirty camera frames per second are inputted into two separate object detection models - a model for common objects, and a model for street signs. If a common object or street sign is detected, a labeled bounding box is drawn around the object on screen. Additionally, the object is enqueued into a priority queue. This priority queue is dequeued at regular intervals as text-to-speech. Street signs are considered more important than other objects, and will be placed further up in the priority queue. Also objects receive priority based on their location on-screen. The collision avoidance module is implemented using ARCore's Depth API, wherein a depth map is created of the environment using a depth-from-motion algorithm - meaning one camera is sufficient to attain accurate depth information. If the phone camera detects a depth less than four feet, a high priority alert is enqueued into the front of the priority queue.

Section 6: Team Member Retrospectives

Selin Onal: By far the biggest time-sink for this project for me was spending so much time trying to implement a navigation system that relied on Google's Map and Navigation APIs. In the end, it turns out Google doesn't let users implement their own version of Google Maps because of copyright concerns. The app turned out just fine without the navigation portion, but if I had anticipated this from the beginning, I could have thought of another module to implement, which could have made our app even better. Other than that, I wish I considered using fragments from the beginning; while it didn't take very long to reimplement the modules as fragments, starting with fragments would have made the code a lot cleaner in the first place, and I would have never needed to implement a navigation system between module activities that shared information across activities that we never ended up using.

Rick Sear: A large portion of my time was spent searching and reading Tensorflow documentation and tutorials, which are numerous and poorly-matched with current Tensorflow versions. Having gone through that process and assembled a step-by-step guide for training an object detector (see §7), I could probably be finished with the sign detection model in about a month. I've also seen where the sign detector works best and where it fails, so I would also want to spend some time augmenting the training dataset with certain sign categories that pose a particular problem for our current model to hopefully get some better performance. Additionally, we spent a lot of time planning for an SSD-to-neural network pipeline which didn't end up being feasible, so I would definitely recommend "Past Rick" skip the neural net business and just go straight to training an SSD model; it ends up being cleaner.

Jack Sloane: If I had to do this project again, I would shift my focus away from combining depth analysis and machine learning into an integrated module that runs at the same time. Instead, I would focus on developing and improving the machine learning and depth analysis modules separately - to make each module as accurate and efficient as possible. A huge portion of my early work was research and trial-and-error revolving around this problem of running machine learning and depth analysis at the same time. Ultimately I concluded that, although not impossible, it would be unrealistic to implement such a feature in the time allotted to us. If I had instead used this time to improve the object detection and collision avoidance modules, perhaps we would have faster and more accurate object detection - and/or a more sophisticated collision avoidance algorithm. I would not, however, change the APIs used for machine learning or depth analysis. Tensorflow Lite is the most powerful and easiest to use API available for computer vision on mobile devices. Also, the depth API for ARCore is perfect for creating a map of the environment containing accurate depth information, and it's designed to be run on any modern Android device.

Section 7: Instructions for Future Development

7.1: Bootstrapping

Follow the installation instructions in our README. The app itself should be very straightforward; just open the “dog-app” folder in Android Studio and it will handle the installs for external libraries. The Tensorflow/Python installation will be a bit trickier, but that part is completely independent of the Android app. Basically, if you only want to work on the app and not change the underlying machine learning, you don’t need to mess with the “model-training” folder.

7.2: Object detection improvements

One major (and relatively straightforward) improvement would be to add additional SSD models. We currently have a “general” object detection one and a “street signs” one, but categories like traffic lights, vehicles, or even faces could deserve their own model. You could also use Mapillary or a different dataset to *re*train our sign detector to make it better.

Beware: as of our Senior Design project, the Tensorflow documentation and tutorials are very muddled between Tensorflow 1 and 2, so make sure you pick a version (we used TF2) and read everything very carefully to ensure it is for or about the correct version. This set of instructions is also included in our README, but the following is our list of instructions for training your own SSD model (assuming you have all the Tensorflow libraries properly installed and a dataset ready to go):

1. Write a preprocessor for your dataset. We have included our Mapillary preprocessor (mapillary_prep_obj.py) for reference. Your preprocessor needs to be able to read the dataset and output .record files (this is a Tensorflow file format) that have the following values for each object:
    a. filename
    b. xmin
    c. ymin
    d. xmax
    e. ymax
    f. class
    g. width
    h. Height

You also need to output a label map file. Hopefully our example file provides a good template structure for how to do this.

2. Once you have train.record, val.record, and test.record, download a model from the Model Zoo. Extract the folder and configure the pipeline.config file to match your data. At the very least, this means changing num_classes and input_path/label_map_path values. You'll probably need to downsize the batch_size as well, depending on your system's RAM. Any bugs you hit while training will probably need to be addressed in this file.
3. Run: python model_main_tf2.py --model_dir <PATH TO NEW MODEL DIRECTORY> --pipeline_config_path <PATH TO pipeline.config>
4. Run: python export_tflite_graph_tf2.py --pipeline_config_path <PATH TO pipeline.config> --trained_checkpoint_dir <PATH TO NEW MODEL DIRECTORY> --output_directory <PATH TO NEW TFLITE MODEL DIRECTORY>
5. Run ("MODEL NAME" should end with .tflite): tflite_convert --output_file <MODEL NAME> --saved_model_dir <PATH TO NEW TFLITE MODEL DIRECTORY>/saved_model
6. Modify lines 10, 12, 14, 15, 41, and 42 in tflite_meta_packer.py. Notably, line 41 should point to the label map file and line 42 should point to "MODEL NAME" from Step 6. Then run: python tflite_meta_packer.py
7. You now have a custom Tensorflow Lite model ready for use in Android.

7.3: More ideas

Here are some other ideas we would want to see in a future version of the app.

- GPS: Can D.O.G. be used as a navigation tool? Integrate the app with Google Maps or a different location service so that it can give directions alongside object information and collision avoidance.
- Text recognition: Make D.O.G. read nearby text and convey this information to the user. This could be as simple as directly reading text printed on signs or other text or as advanced as using NLP to glean information from these sources to deliver more directed narration.
- Merge our modules: Bring our collision avoidance and object detection modules together into one unified stream of cues. This task is harder than it might sound, requiring an advanced knowledge of Android Fragments, Android's resource management, and a clever plan to allow multiple high-resource services to access the camera simultaneously or close to simultaneously. The ultimate realization of this goal would be for D.O.G. to be able to announce position *and* distance data for objects, e.g. "Car 5 feet to your right" or "Tree 7 feet ahead".
- Outreach: We weren't able to make contact with the American Foundation for the Blind, but outreach to our target audience would be crucial to making this app viable. A large foundation like AFB could be a potential funding source, and feedback from focus groups

of low-sighted or blind people is, in our opinion, a necessary step before releasing this app. Our goal with D.O.G. was to make an app that’s free or nearly free as an alternative to some very expensive tech that currently exists. We want this app to be both informed by its user base and accessible to them.