Andy Metcalf

**Optimal Solutions Internship Report**

**Device Description and Development Details**

Out of the box, the Enterprise Edition Google Glass comes with no accessory applications. It has a camera that is capable of taking 5MP photos and 720p Videos. The Glass supports both wifi and bluetooth connections. The device has and Intel Tangier Dual core processor with 16GB of storage and 2GB of RAM. The operating system is Android 4.4 Kitkat, which corresponds to API level 19.

Developer control of the device is performed through Android Debug Bridge (ADB). ADB is used to control basic settings, such as the devices time and wifi connection, and to manage applications installed on Google Glass. We were given access to the [Glass Enterprise Partner Development Program](https://sites.google.com/a/google.com/glass-partner-dev-program/getting-started) website, which among other things, specifies the necessary ADB commands that are useful to control the device. [ADB shell](http://adbshell.com/) is a website that was also helpful to determine specific ADB commands. Android Studio was used as the environment to develop all applications.

**Application Development**

Three applications were developed over the summer with specific use cases for the client (JLG) in mind. A demo app was made that demonstrates the general capabilities of Google Glass, as well as both object classification and object detection apps to be used in tandem with the Glass camera for image processing.

**Demo Application**

The demo app includes timer/checklist, camera, and voice recognition demonstrations. The timer/checklist utilizes both touch gestures and voice commands to walk through ten tasks, labeled “task 1 - 10” for generality. At the start of each task, the user can either tap or say “START TASK” to begin the task, at which point a timer is started that continues until the user either taps the device or says “STOP TASK”. The user then proceeds to the next task with voice or touch commands, and the individual task times are displayed in a list on the right of the Glass screen. This app was created with JLG in mind, since they perform many tasks that need to be timed and would like to automate their processes with hands free operation.

The application was tested onsite at JLG during several of their machine tests. The voice recognition feature of the app was determined inadequate to meet JLG’s standards due to the inaccuracies that it introduced to the timing process. In order to activate voice recognition on Glass, the user needs to state “OKAY GLASS”, followed by their specific command. For example, to stop the timer, the user would have to say “OKAY GLASS” > “STOP TASK”. The amount of time that the user spends dictating these oral commands introduces error into the timing process that was outside of JLGs tolerances. Utilization of the app with touch gestures was found to be more accurate, but results in the loss of a hands-free user interface. Google glass also implements blink gestures which could be utilized in the future to continue to provide a hands-free user interface.

**Object Classification Application**

The classification app allows the user to take a photo with Glass and returns that photo with an annotation stating what the primary object in the image is. The app utilizes wifi socket connections to communicate with a server running a machine learning algorithm on the photo to produce the annotated result.

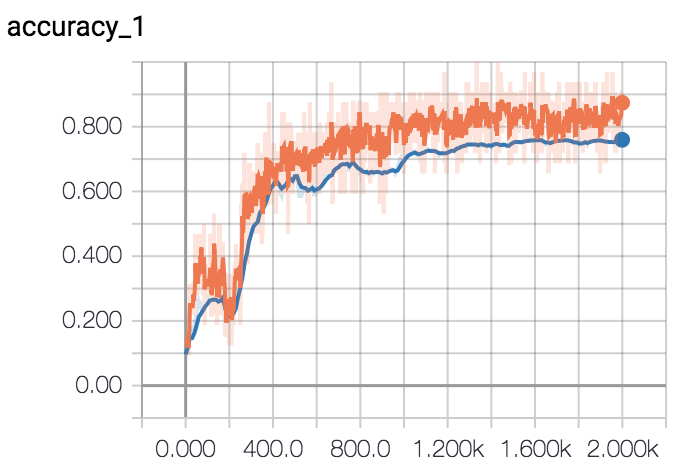
**Data Collection**

Numerous images were taken of specific machine types from various angles and distances for the JLG machine classification task. The specific machine types and number of images are listed in the table below:

|  |  |
| --- | --- |
| JLG Machine Type | Number of images taken |
| 340AJ | 145 |
| 450AJ | 123 |
| 460SJ | 154 |
| 460SJC | 109 |
| 600S | 38 |
| 800AJ | 116 |

**Training**

The InceptionV3 convolutional neural network architecture was chosen for the machine classification task. The network was trained for 2000 epochs on a Tesla K80 GPU using a batch size of 32, data transformations and a learning rate of 0.0001. Total training time was 100 minutes. The network was able to achieve approximately 76% accuracy on the Validation data set.



A portion of the error for this network could be a result of the training images themselves. Many of the images are not of isolated machines because JLG lines up their machines next to each other in the stockyard, making it impossible to take a picture of a machine in isolation. Additionally, many of the images do not include the entire machine because the machines are too big to fit in the frame. Perhaps some image preprocessing and network parameter tweaks could improve network performance. More data may also improve network performance. This pretrained network was not included in the classification app.

The classification application that was produced for the Google Glass is more general in nature. It also uses an InceptionV3 CNN, but instead uses weights trained on the [ImageNet](http://www.image-net.org/) data set. This generality, allows the user to test the app in many more environments since the classes are more general than JLG machines.

The application allows the user to take a photo which is sent via socket wifi connection to a computer that is set up as a server. That server executes a python script which performs image classification on the input photo and returns the same photo with an annotation stating what the primary object in the photograph is. This new annotated image is then sent from the server back to Google Glass where the user can visualize the result. There is significant latency associated with this application; mostly due to the python script implementation. After the server accepts the photo from the Google Glass, it must execute a python script which requires several modules to be loaded into memory before it can run the algorithmic code. This loading process takes time and results in application latency. We could avoid this latency in the future by implementing the machine learning in Java - the server’s programming language, or by starting the python script earlier to load the modules before the image comes in. The total inference time and python script implementation time are listed below:

|  |  |
| --- | --- |
| Total inference time (avg of 10 trials) | 34.6 seconds |
| Python script implementation time (avg of 10 trials) | 30.4 seconds (88% of total time) |

**Object Detection Application**

The object detection application allows the user to take a photo with the Glass and have that photo returned to the Glass with labeled bounding boxes around objects that the network has been trained to identify. It again utilizes wifi socket connections to communicate with a server running a machine learning algorithm on the image to produce the annotated result.

**Data Collection**

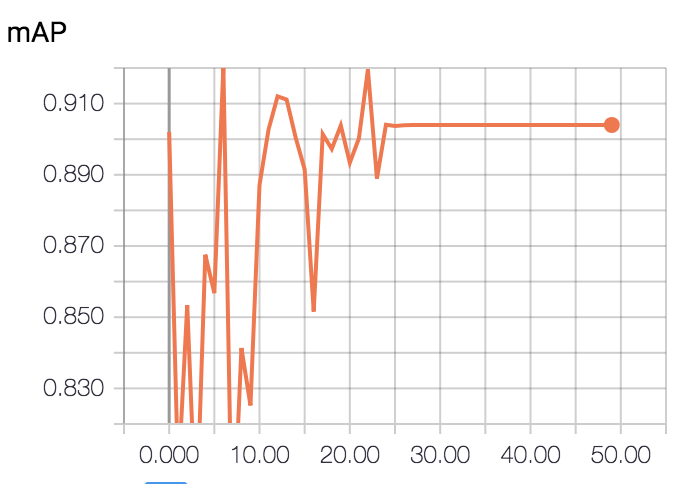
The specific use case for JLG was to detect blemishes on their machines and to determine blemish size. Ripped pieces of masking tape of specified lengths were used to represent blemishes and were placed on the machines at various locations. Five or more pictures of each piece of tape were taken from different angles at specified distances (1 foot and 5 foot). The amount of data, split into training and validation sets is listed in the table below:

|  |  |  |
| --- | --- | --- |
|  | 1 Foot | 5 Foot |
| Training | 619 | 1140 (total - not separated) |
| Validation | 264 | 1140 (total - not separated) |

A tool called [labelImg](https://github.com/tzutalin/labelImg) was used to label the collected data. It was used to draw bounding boxes around the blemishes and label them as such. A XML file was created to store the bounding box coordinates and labels for each image. The picture files and XML files were then stored in [Pascal VOC](http://host.robots.ox.ac.uk/pascal/VOC/) data format to be used by several machine learning algorithms.

**Training the Network**

Several algorithms were used to train the blemish detection network. Due to time constraints, training was only performed on the images taken at a 1 foot distance. The Tensorflow API was used to implement both [SSD](https://arxiv.org/abs/1512.02325) and [Faster RCNN](https://arxiv.org/abs/1506.01497), achieving fair results with SSD (65% mAP), and excellent results with Faster RCNN (97% mAP). However, these results were not analysed thoroughly since it was soon discovered that the Tensorflow API was too much of a black box to extract the necessary information for our research. (We needed the filenames to extract the information necessary for future work with blemish size detection). Keras was chosen for algorithm implementation since it is a Tensorflow wrapper that makes coding Tensorflow much more user friendly. A state of the art algorithm called [Retinanet](https://github.com/fizyr/keras-retinanet) was chosen to to perform object detection on our blemish images. The first Rentinanet blemish network was trained for 50 epochs and took approximately 7 hours to train. The network achieved an 85% validation mAP with a confidence = 0.05. The confidence parameter relates to how confident the network is that the bounding box contains a blemish. A confidence of 0.05 implies that the network will look at every bounding box that it thinks has a 5% chance of containing a blemish. This is a low confidence setting, so we increased the confidence level to 0.60. It would be expected that the network would achieve higher accuracy at the higher confidence since more of the bounding boxes should contain blemishes. To our surprise, when the confidence was raised, we found that the accuracy dropped to 77% validation mAP. This result seems to imply that the network was confused and needed more data. Thus we decided to retrain the network, applying random transformations to the images and running through the entire training set four times per epoch. This improved results, yielding a validation mAP = 90% (at confidence = 0.60). Validation accuracies per epoch for this second network are shown in the figure below:



Inference time per image was 0.91 seconds on the GPU (fist network) and 3.82 seconds per image on the CPU (second network).

**Application Implementation**

The object detection app can toggle between a general network derived from the pretrained Retinanet weights that is capable of detecting 100 common objects, and the trained blemish detection network. The app allows the user to take a photo with the Google Glass and it is sent to the computer/server is the same fashion as the classification app described above. Again, the time constraint occurs when the python script is executed by the server. The annotated photo is finally returned to the user with labeled bounding boxes around the objects it recognizes; be it common objects in the general network, or blemishes in the trained network. The total inference time and python script implementation time are listed below:

|  |  |
| --- | --- |
| Total inference time (general) | 16.7 |
| Python script implementation time (general) | 11.6 (69% of total time) |
| Total inference time (blemish) | 21.7 |
| Python script implementation time (blemish) | 16.8 (77% of total time) |

**Future Blemish Size Detection**

The next step with this application is to be able to determine the size of the blemish. A first pass could be completed by computing the blemish size with the size of the bounding box. This would likely introduce considerable error, since photos are taken at different angles and the blemishes are placed at various angles, causing the bounding boxes to vary in size. It however could be used as a baseline to measure the performance of more sophisticated methods. Those methods could include a regression sub-net added to the Retinanet network, or a seperate CNN applied to the individual blemishes detected by Retinanet.

**JLG Trip**

In this section, we briefly discuss the work and testing performed onsite at JLG.

**Data Collection**

In addition to the machine and blemish images that were collected (for which the data collection process is described above), noise data was also collected. Voice samples were collected at both “high noise” and “low noise”. These noise levels correspond to standing next to an open engine cover with a machine running at the highest RPM, and just standing in the testing yard with passive engine noise.

**Device Testing**

At JLG, the device was tested in a working environment. The voice recognition features were tested while standing next to an open engine compartment at the “high noise” level. At this decibel, with the machine running at maximum RPM, the Glass was always able to detect voice commands. There was also direct sunlight and relatively high temperatures (around 90 degrees fahrenheit as I recall) so it was a good environment to determine if the device would be susceptible to overheating (as it had been found to do in previous experimentation). During our testing, no device experienced overheating, though one defective device did seem to require hard booting at a greater frequency. As mentioned above, the timer app was also tested, resulting in the realization that voice gestures that require “OKAY GLASS” to be activated introduce an error that will be unacceptable for JLG requirements.

**Device Usefulness for JLG**

JLG employees were intrigued by the Google Glass. The ability of perform hands-free data entry in an automated, consistent fashion could potentially yield gains in both efficiency and quality control. JLG employees performs many checklist type tasks with data entry often being performed on a clipboard. These notes are then be manually entered into a database - a process which is both time consuming and could easily result in information being lost in translation. Google Glass has the potential of automating these types of checklist procedures. Several more advanced use cases were also discussed, including blemish size detection and automated drift test measurement that would implement machine learning techniques.

**Constraints for JLG**

Complementing their excitement for the device were several concerns. Will the device be easily transferable between workers? What if a worker has prescription glasses? Can Google Glass be worn in conjunction with safety glasses? How durable are the glasses? Can they withstand temperature extremes and precipitation? Can the glasses be worn with various hat types? Will some hat types cause the touchpad to be covered? Will the touchpad work with gloves?

We were not able to get a well defined answer relating to durability with regard to temperature and weather extremes. The touchpad does seem to work with gloves that are fairly thin, but will not work with thick winter gloves. The hat issue seems like it could become problematic in winter months when a hat would likely be covering the ears. We concluded that the Glass does not work in conjunction with all safety glasses. I was able to fit a Glass without lenses over my sunglasses, but the screen was then shifted out of viewing range.

**Device Defects**

**Overheating**

While testing and developing apps for the Google Glass, we experienced issues with overheating several times. The device was found to overheat in the direct sunlight (with temperatures around 100 degrees Fahrenheit) when using the camera app. It was also found to overheat indoors while testing computationally expensive applications. Several applications, such as the sideloaded barcode apps, that made heavy use of the camera, caused the glass to overheat. Additionally, while developing apps, I found that long *while loops* occasionally caused the device to overheat.

**Rebooting**

Both devices I have tested have experienced crashes that require the device to be hard-booted. This process is easily done by holding down the power button for at least 15 seconds, followed by a quick push. This happens fairly frequently and is perhaps associated with USB connected app development, but more testing would be necessary to determine the exact frequency and cause.

**Device Difficulties**

**WPA2 Enterprise Wifi Connection**

The wifi at UVM is a secured WPA2 Enterprise network, which is a common security protocol for large institutions. I have submitted two tickets to Google and gone into the UVM tech department twice trying to connect Glass to the network with no success. This is a problem for my ML apps since they communicate over a wifi network. So far I have only been able to connect the glass to WPA password protected networks. My correspondence with Google will be included in the GitHub code page for future reference.

**API Level**

The current operating system is Android 4.4 Kitkat, which corresponds to API level 19. This API is quite outdated and is not compatible with many of the Android based machine learning libraries that are now available. If the API were updated, we could perform inference on the device itself - a process that would likely reduce inference time considerably.

**“OKAY GLASS”**

As described above, the voice recognition feature was not acceptable for the JLG timer since the user needs to say “OKAY GLASS” to activate the feature. More research is necessary to determine if we can bypass this requirement in certain apps so the user could simply state “START” and “STOP”.

**Barcode Detection**

Barcode detection is a feature that should be able to be accomplished on Google Glass. The capability is natively built into the Glass to detect wifi QR codes, so it is indeed possible. I was however unable to find or build a successful implementation. I tried sideloading several Android apps as well as adapting some of the existing code on GitHub. I’m quite positive that with more time, a barcode reader could be implemented.

**Older Code on Github**

There is quite a bit of code on GitHub for Glass development, however, because the original consumer based Glass was a flop, the codebase is quite old. Most of the repositories have not been updated since 2014 or 2015 and the code is often quite depreciated.