NAU NORTHERN ARIZONA UNIVERSITY
School of Informatics, Computing, and Cyber Systems

To: Robert Severinghaus
From: Noah Levie, Daniel Copley, Mohammed Almershed
Date: March 26, 2021
Subject: Testing Results Report

Our group was tasked with the goal of designing and creating an autonomous structural inspection drone. Presently, building inspection is a lengthy and manual process. Furthermore, it can be dangerous. It requires trained inspectors to climb ladders to inspect exterior features roofing, windows, and other hard to reach items. We aim to solve this problem by creating an unmanned aerial system to do this automatically. With a few strokes by an operator, the drone will launch itself, gather and process footage, and report back with broken or damaged building features. We utilized open source technologies such as the Pixhawk Flight Controller, PX4 Firmware Stack, QGroundControl, YOLO object recognition, TensorFlow, and OpenCV, in combination with our own custom software stack to enable our structural inspection drone solution.

We performed five documented tests, three of which were step-by-step unit tests, one was a matrix unit test, and one was an integration test. We also performed a variety of undocumented inspection tests on features such as the drone assembly and the drones ability to interface with the ground control station. The first test we conducted was verification of our object detection machine learning model. We conducted 12 tests, two on each type of feature we want to detect (ie. window, gutter, etc). For each feature, we tested both the damaged and undamaged cases to make sure the detection was working correctly. This test took around 28 hours in total, including the time it took to train the models. Results were mixed but mostly positive. Our mean average precision was 78%, 2% shy of our 80% goal. The second test we performed was verification of our damage severity classifiers. Again, we tested both the damaged and undamaged cases for each classifier to ensure proper function. The classifier networks train much faster, as they are intended to be lightweight. We spent around 2 hours completing this test. The classifier correctly predicted the result for each of our cases, however, the mean average precision was lower than we hoped for. The third test we conducted was on our action generation. We conducted 8 tests on our action generation. We spent approximately 1 hours, testing the action generation in the flight control simulator to ensure we hadn't overlooked any edge cases which might produce a dangerous output. All of our action generation tests passed successfully. The fourth test we performed was to determine the best deep neural network configurations for each of our classifiers. This test trained 27 models for each of the classes and compared their efficacy. Training and testing took around 7 hours. This test was very successful, and helped us choose our network configuration. The final test we conducted was our system integration test from feature detection to drone action. For this test we input a variety of features to observe the

generated response. This test was largely successful, however, a few mistakes in the image processing pipeline led to abnormal responses.

EE486C Capstone
Team 2
Structural Monitoring Drone
26 March 2021

Introduction

Our group, Team Skeyes, is working with Dr. Abolfazl Razi, an engineering professor at Northern Arizona University in the Wireless Networking and Smart Health (WiNeSH) research laboratory. His personal projects are centered around predictive modeling for different applications including Wireless Networking, Smart Cities, IoT and Aerial Systems. His goal is to design new machine learning tools that model and predict network status change, user behavioral trends, and traffic mobility in order to accommodate predictable events by taking early decisions. He also does work in the medical engineering field, involving the development for tools for predictive modeling of biomedical signals for smart health applications. All of his projects are supported by the NSF, NIH U54, US Airforce Research Laboratory, and Arizona Board of Regents (ABOR). A link to his personal website and portfolio can be found on the NAU website at https://www.cefns.nau.edu/~ar2843/.

Conventional approaches to building inspection are laborious, costly, and dangerous. As it stands, an inspector has to personally travel around any building he would like to inspect; furthermore, in the case of a particularly tall building, he would have to be elevated to thoroughly survey the external surfaces of the structure. This process is slow, expensive, and potentially dangerous. On the contrary, the use of an automated drone for such a task would be faster, cheaper, safer, and more convenient for everyone involved. The drone will allow a user to plug a device into their laptop and give the program some basic instructions, then receive a streamlined output of video data that they can use in lieu of performing a manual inspection. The user will not have to physically move in order to investigate the building for flaws, they can stay in one place and observe the process from a comfortable or convenient location. This drone additionally makes the inspection of taller buildings more feasible, as the operator will not have to elevate themselves in order to achieve a close investigation of the outer surfaces of the building. Additionally, any data that is recorded by the drone can be conveniently stored and later accessed by the user. Ultimately, our goal with this product is to reduce the risk, inconvenience, and potential overhead of having to perform a relatively simple task such as a building inspection.

The design of our project is heavily reliant on both hardware and software components. The hardware is primarily separated into two fields: the drone apparatus and the ground control station. The drone is a quadcopter UAV, which is outfitted with a number of sensors and transmitters/receivers. The hardware system architecture can be seen in Appendix B**.** Each of the four motors is controlled by an electronic speed controller. All of the power for the drone and auxiliary equipment is supplied by a lithium-polymer battery, and it is regulated and distributed by a power distribution board. The flight controller mounted on the drone receives positional data from its GPS antenna, accelerometer, gyroscope, and magnetometer, and it transmits this data to the ground control station via its telemetry radio. The drone carries a radio control receiver to be manually controlled by a remote controller. A gimbal system is mounted on the

drone, carrying a GoPro Hero 4 camera, which streams analog video signals via radio back to a receiver connected to the ground station. The ground station consists of a laptop running the QGroundControl flight control software, connected to a telemetry radio for sending mission commands as well as a radio receiver for video data. The machine learning object recognition systems are implemented within the ground control software stack on the ground control station.

The two primary components of our software are the YOLOv4 object recognition system and the severity classification models, which are implemented using TensorFlow and OpenCV in Python. The overall structure of our software follows the model-view-controller (MVC) architecture, which is a paradigm used to implement user interfaces by separating the project into modular components and smoothly controlling and standardizing the flow of data through the system as represented in Appendix C. This data flow will occur within the ground control station laptop, which receives the video data wirelessly from the drone, and can be seen in Appendix A. This video data is sent to our YOLOv4 model, which annotates the video feed and displays it to the user in QGroundControl. At the same time, it outputs snipped images of structural features to the damage classification network using OpenCV. This classification process will determine what inputs are prompted from the user, i.e., whether they want to continue the mission or pause to keep inspecting the feature. These commands will be transmitted to the drone via MAVLink and will be displayed to the user in QGroundControl.

System Architecture

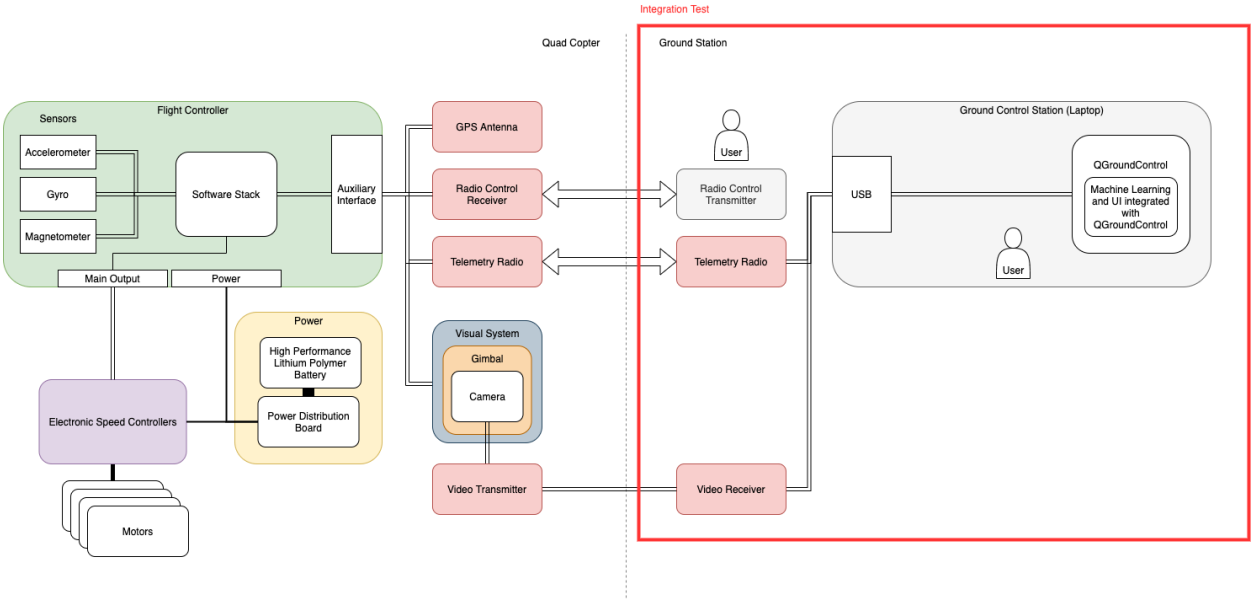


Figure 1. Hardware System Architecture with Tests

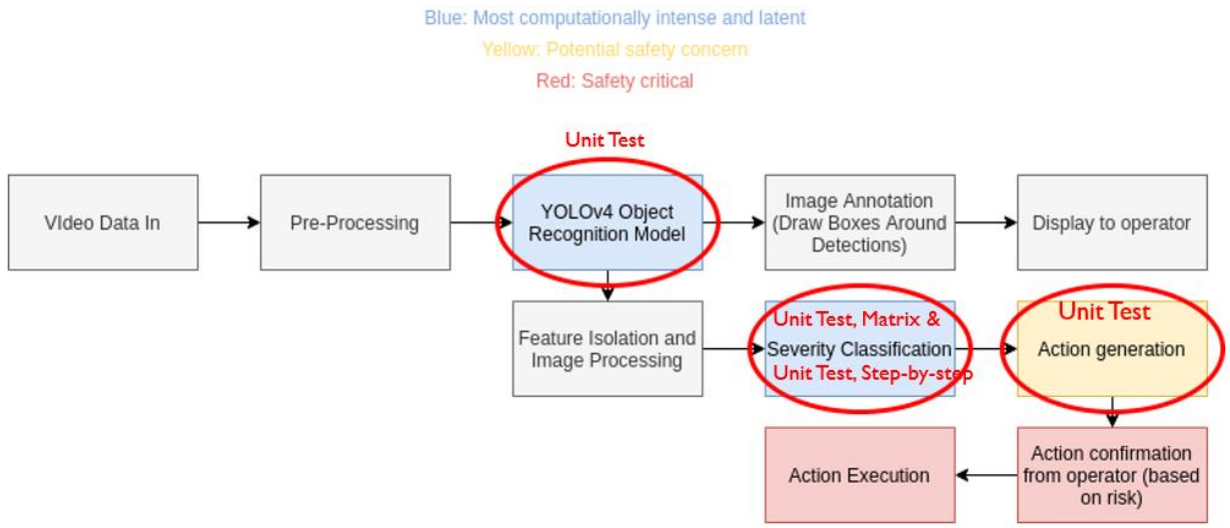


Figure 2. Software System Architecture with Tests

Requirements Spreadsheet

Type of Test	Status	Req #	Requirement
		1	**User Interface**
Inspection	PASS	1.1*	The drone must be operable using a graphical user interface
		1.1.1	Allows the operator to navigate the drone, configure inspection or navigation settings, or access data from the drone's flight.
		1.1.2	This should be written in a common programming language such as Python or MATLAB
Inspection	PASS	1.2	The drone should be operable and configurable wirelessly
Inspection	PASS	1.3	The user will be able to see both the drone's positional data and video stream during its flight, and be able to issue commands mid-mission
		2	**Image Processing**
Integration	FAIL	2.1*	The drone will use image processing in order to isolate and identify features relevant to the operator, which should be at least 80% accurate
Inspection	FAIL	2.1.1	There will be four recognizable features relevant to our project: gutters, windows, walls, and roofs
		2.2	Each of the structural features will be initially identified by a YOLOv4 neural network, with a reasonable degree of speed and accuracy
Step-by-step	FAIL	2.2.1*	Each YOLOv4 feature should be recognizable with at least 90% accuracy
		2.2.2	The YOLOv4 system should output appropriately cropped images to be classified for damage severity
Inspection	PASS	2.2.3	In order to accomplish this, the team will collect at least 300 images for each category
		2.3	A variety of classification neural networks will be run on the detected features, specialized for each feature
Step-by-step	FAIL	2.3.1	The damage classifiers should be at least 90% accurate in distinguishing between faulty and intact features
Step-by-step	PASS	2.4*	Feature detection within the video feed will trigger appropriate commands to be sent via MAVLink
		2.4.1	The YOLOv4 object detection event will instruct the drone to halt its mission in order to investigate
		2.4.2	If the feature is classified as defective, the drone will prompt the user to give input, otherwise, the mission will continue
		3	**Drone Assembly**
		3.1	LiPo battery capable of powering drone flight controller, motors, and auxiliary devices
		3.2	Maneuverable camera mount, capable of 3 degrees of freedom
		3.3	Flight control systems should strive to communicate over a single unified protocol
Inspection	PASS	3.4	Power should be supplied to all auxiliary components via the same power distribution board
		3.5	Drone will stream analog video data in real time to the ground control station
		4	**Flight Path Navigation**
	PASS	4.1	The drone must be operable using a remote controller
	PASS	4.2	The drone must be capable of semi-autonomous flight (not directly navigated by remote control)
	PASS	4.2.1	The mission flight mode will receive and quickly react to events in the form of MAVLink instructions
	PASS	4.2.2	The drone is capable of safe takeoff/landing protocols
		4.2.3	The drone's mission flight mode allows for the designation of regions of interest for surveying

Figure 1. Requirements, Status, and Test Types

Primary Requirements

Requirement 1.1 – The drone must be operable using a user interface. The purpose of the graphical user interface is to configure the drone upon start-up, as well as to ensure a smooth display of information and prompts to the user. This requirement is important to our client because he would like the drone to be theoretically operable by an individual who is not an engineer or software designer. It should be clearly laid-out and easily operable by reading the user manual, which will detail the recommended settings and parameters of the drone. This requirement is a core feature of our project, and if it were not met then this drone would never (theoretically) go into production or distribution, since it would be largely inoperable by the general populace.

Requirement 2.1 – The drone will use image processing in order to isolate and identify features relevant to the operator, which should be at least 80% accurate. The system necessarily will use image processing and feature detection in order to semi-automate the inspection process. In order to ensure that the drone would actually be effective in practice, we have set a baseline accuracy of 80% when both the object recognition and damage classification systems are implemented. This requirement is important to our client because his field of study largely involves image processing and machine learning applications. If the device did not effectively implement machine learning, it would simply be a video-recording drone, which is not particularly innovative or revolutionary. If we were to fall short of the accuracy threshold, the project would not be ruined, as long as we were able to provide proof of concept; however, we would not want to deploy the device in a real-world setting.

2.2.1* Each YOLOv4 feature should be recognizable with at least 90% accuracy. This requirement indicates that the You Only Look Once v4 (YOLOv4) object recognition system should have a mean average precision (mAP) of 90%. This means that the when the average precisions of all the classes are averaged, the result should be 90%. This is important to our client because it is a central feature of the drone as an inspection and analysis device. If this threshold were not met, once again, the project would not be ruined, but we would not want to deploy the device in a real-world setting.

Requirement 2.4 – Feature detection within the video feed will trigger appropriate commands to be sent via MAVLink. This requirement means that the YOLOv4 object detection event will instruct the drone to halt its mission in order to investigate; if the feature is classified as defective, the drone will prompt the user to give input, otherwise, the mission will continue. This feature is additionally important to our client in order to distinguish it from a simple video recording drone. The drone should be able to react dynamically to the visual stimuli that it is receiving, otherwise the drone is fairly mundane, and the project is ultimately a failure.

Testing Types

UTM stands for Unit Test, Matrix, which is a form of testing in which a series of pre-determined, varied inputs which are structurally the same are given to a system to test the function of a single process. We performed this test to identify the configurations that were the most effective for each of our damage classification convolutional neural networks. This was useful because we were able to test a range of convolutional layers, dense layers, and nodes per layer in rapid succession, which demonstrated to us the most optimized format for different datasets.

UTS stands for Unit Test, Step-by-step, which is a form of testing in which a series of instructions are performed within a system, perhaps with a single input, which are completed in a number of stages. We primarily used this type of testing since it is the most relevant to the evaluation of the performance of neural networks. For both our object recognition and damage classification systems, we were able to input a validation dataset, as well as individual images to be processed and displayed, in order to calculate the average precision (AP) of each using the built-in functionalities of Darknet and TensorFlow, respectively.

Integration Testing is used to ensure that multiple subsystems within the product function nominally together, as opposed to the performance of individual modules. We used this to test the latency, fidelity, and overall effectiveness of the data flow through our system. There are a number of different software components which are receiving and outputting data, so this method allowed us to effectively analyze whether the individual components were working properly together. For example, the camera takes a video, which is transmitted to the ground control station receiver then analyzed by the object recognition system, cropped, and sent to its respective classification network. At this point it is displayed to the user and a particular action or prompt is generated based on the severity of the damage.

Inspection Testing is a type of testing in which a simple visual (or other sensory) assessment of a component is performed, without any specific testing process or inputs. We primarily use this type of test to ensure that the assembly of the drone is nominal. For example, we can inspect the drone to make sure that the components are all attached tightly, that the drone is receiving input from the remote controller and flight control software, or that the video data is being collected by the camera then transmitted and received between the drone and the ground control receiver.

Major Documented Tests

Major Test 1: YOLOv4 Object Recognition

The first major test that we performed was the YOLOv4 object recognition accuracy test. We trained the YOLO model three times, using varying numbers of classes and sizes of datasets. The process was performed using a Jupyter notebook in Google Colaboratory, which allows users to access a graphics processing unit (GPU) via the use of a virtual machine. These models each took over 8 hours to train, even with the GPU acceleration, so we generally trained them overnight.

The first model that we trained was only trained to recognize windows. The image dataset was fairly low – at around 150 images – as it was our initial test on learning how to train a neural network. It was fairly accurate, considering its small training dataset, with an average precision (AP) of 73%, since it was only set to recognize windows.

The second model that we trained was trained to recognize windows, gutters, roofs, and walls, and was quite inaccurate. The dataset for the walls and roofs was practically nonexistent, being around 25 images each, and this model took much longer to train due to having 4 classes to attempt to recognize. The windows and gutters had about 200 and 100 images, respectively, in their training sets, and had a mean AP of around 70%, which we believe was somewhat reduced by the attempts to identify other classes.

The third model that we trained was only trained to recognize windows and gutters, since we had the most data for those two sets – 300 and 100, respectively – and they have the most recognizable features. This was significantly quicker to train than the previous, since there were half as many features, and it was more accurate as well. The mean average precision was 78% with this model, with an AP of 72% for gutters and 84% for windows. This model was more reasonably functional and served well as a proof of concept since it could reliably find windows, even with relatively low confidence values.

Major Test 2: Damage Severity Classifier Optimization

This test was a black box matrix unit test designed to help us determine the most effective configuration for our convolutional neural networks (CNNs). CNNs are fundamentally made up of convolutional layers and dense layers, which are implemented with different activation protocols, different numbers of nodes, and varying levels of compression. The only way to effectively design a CNN is by trial and error since it cannot necessarily be calculated based on the image dataset available. As such we divided our image datasets for windows and gutters into defective and nominal categories for training. Using a series of nested for-loops, we configured and trained 27 different models on the same image datasets. We used an image input size of 128x128 pixels, normalizing the training and testing data, as necessary. We used a batch size of 8, due to relatively small datasets of 300 windows and 150 gutters, and 15 epochs to avoid overfitting, as we have found most effective previously. We performed this process on both of our datasets, allowing us to find that the model with 3 convolutional layers, 32 nodes per layer, and 2 dense layers was the most effective for our window dataset with a validation loss of 0.4803 and a validation accuracy of 80.6%, as seen in the table below. Similarly, when run on the gutter dataset, we found that a CNN with 3 convolutional layers, 128 nodes per layer, and 0 dense layers was the most effective with a validation loss of 0.5007 and validation accuracy of 70.73%, for a mean average precision of 75%. This was below the 90% mAP threshold that we were attempting to achieve; however, once again, it was accurate enough to be functional for the purposes of testing the integration of all of the components together.

	Input			Results
Test	Number of Convolutional Layers	Number of Nodes Per Laye	Number of Dense Layers	Validation Loss - Validation Accuracy
1	1	32	0	1.2203 - 0.6511
2	2	32	0	0.6418 - 0.6889
3	3	32	0	0.5105 - 0.7642
4	1	64	0	1.3210 - 0.6410
5	2	64	0	0.6124 - 0.7161
6	3	64	0	0.5116 - 0.7724
7	1	128	0	1.4319 - 0.6415
8	2	128	0	0.6958 - 0.6288
9	3	128	0	0.6866 - 0.7345
10	1	32	1	0.6932 - 0.4973
11	2	32	1	0.6932 - 0.4973
12	3	32	1	0.5162 - 0.7703
13	1	64	1	0.6931 - 0.5029
14	2	64	1	1.2960 - 0.6976
15	3	64	1	0.5247 - 0.7782
16	1	128	1	1.4192 - 0.6581
17	2	128	1	0.7285 - 0.5406
18	3	128	1	0.4830 - 0.8060
19	1	32	2	1.0445 - 0.6843
20	2	32	2	0.5616 - 0.7500
21	3	32	2	0.4792 - 0.7831
22	1	64	2	1.3918 - 0.6716
23	2	64	2	0.7490 - 0.7324
24	3	64	2	0.5257 - 0.7972
25	1	128	2	1.3729 - 0.6811
26	2	128	2	1.2811 - 0.6729
27	3	128	2	0.5495 - 0.7897

Results Analysis

All of the basic functionalities were completely effective under the test conditions. The core of our software design, primarily developed by Daniel, was very modular and organized, which allowed us to efficiently design each file systematically and cleanly. The overall implementation worked perfectly well, as did the general flow of image data since the high-level wrapper code ensures that all of the inputs and outputs are normalized and in the proper format. The video input stream, YOLO output cropping function, feature isolation and image filtering, annotated display, and action generation all work completely nominally. We were able to store and implement the YOLO and TensorFlow models and weights files and retrain them as needed without having to change the wrapper code. In a controlled environment, with high quality images and videos, the system worked fairly well, bounded only by the accuracy and reliability of our neural networks. We were somewhat concerned since the system did not seem to function as well on raw data that we collected using the drone flights – data that was less curated. Ultimately, the YOLOv4 network had a mean average precision of about 79% and the classifier networks had a mean average precision of about 72%, which is only about a 57% overall precision.

The results were relatively close to what we expected. We were disappointed to find that the neural network accuracies were lower than we had originally intended them to be. We had hoped to achieve at least an 80% overall precision, which we were unable to do with such limited datasets. Two of the primary requirements were not met in their entirety – the overall precision and the object recognition precision – since we did not achieve the thresholds that we were aiming for; however, we were able to get close enough to create a functioning system, regardless of accuracy or correctness. The graphical user interface and action generation requirements were met.

Lessons Learned

Throughout the process of developing our structural inspection drone, we had a lot of things go wrong. During our first unmanned test flight of the drone, an assembly oversight led to the GPS antenna coming loose and falling out of position. As a result, we hit the kill switch to keep the drone from aimlessly flying away. It then fell from the sky and broke. On another occasion, we faced a similar issue where a propeller became unfixed during the flight, resulting in the drone falling to the earth. These setbacks were unfortunate and inconvenient, but we rebuilt and utilized lock tight and other methods to prevent more mechanical failures from happening.

One of the biggest challenges for this project was developing the machine learning models. For all of us, this was our first real time using machine learning for image processing, so it was an area of huge growth for us. We got our models working pretty well, however, without access to vast quantities of good image data, it wasn't possible to get it working as well as it could have. We gathered our own dataset utilizing a mix of images gathered around flagstaff as well as the internet. Finding high quality images of broken or damaged items was very difficult. Many of them were cartoon drawings or otherwise not useful for our set. Additionally, we weren't able to collect many images of broken features around Flagstaff because people usually fix broken things. In the future, the dataset is definitely one of the areas in need of improvement. Doing so would vastly improve the quality of our object detection and image classification, and furthermore the performance of the entire system.

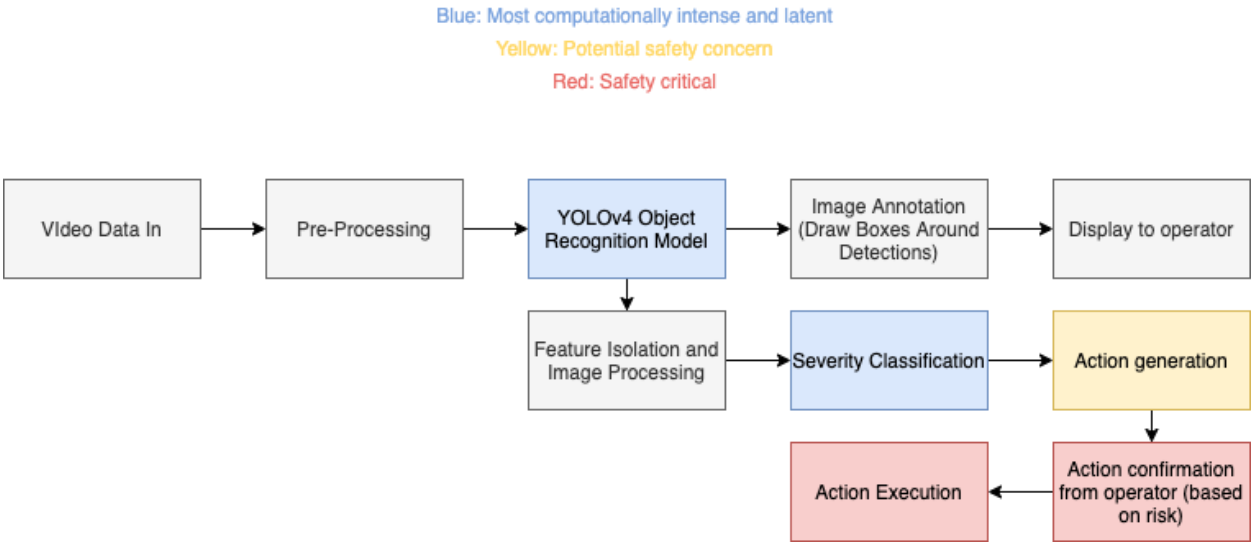
Our tests for the most part were well written and strictly defined. We had no real difficulty testing our system according to our tests. One of the particularly frustrating aspects of our tests, however, was that several of them were on our machine learning models. The machine learning tests were so frustrating for two reasons. Firstly, if it isn’t behaving the way you expect, there is no real way to know why. It is simply not possible to look at the model and understand the decisions it makes. As a result, if something is wrong, fixing it is largely guess and check. Secondly, creating a new model takes a lot of time. Training our object detection network took in the order of days.

Regression testing was done at intervals along the way. We wrote software unit tests to test our software automatically, ensuring that even after a new change was made, the previously tested code still worked. On several occasions, we updated our codebase and introduced new bugs that might not have been otherwise detected. Regression testing is especially important when you are operating a potentially dangerous piece of equipment, such as a quadcopter.

Appendices

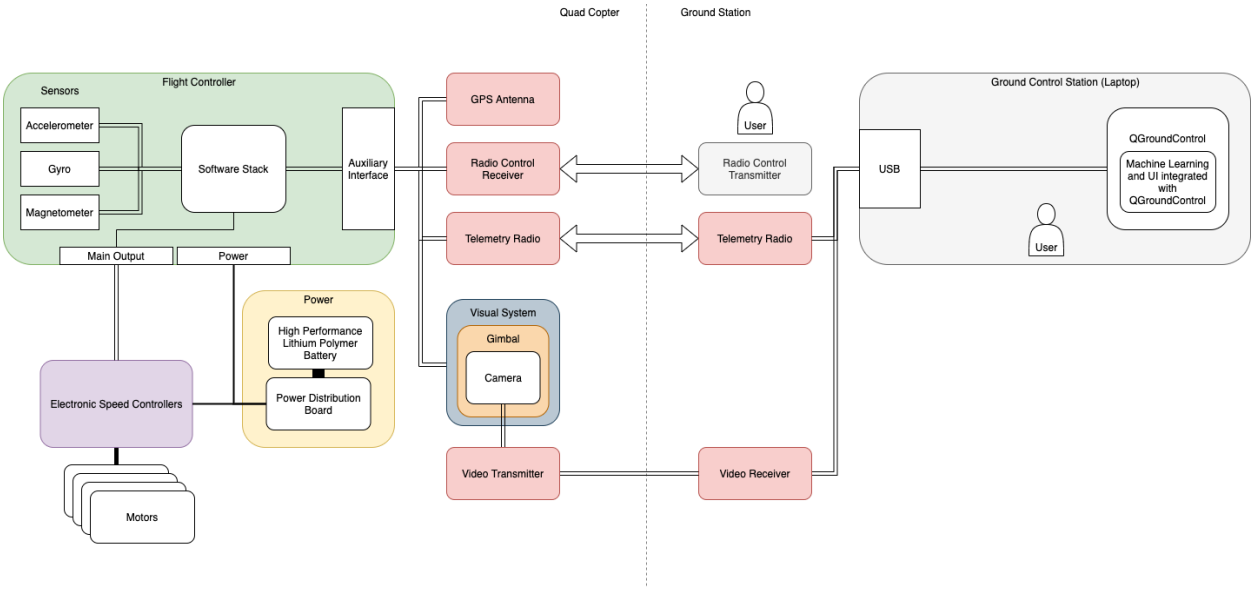
Appendix A – Software System Architecture

This diagram shows the flow of data between software blocks from video input to action output and user display.



Appendix B – Hardware System Architecture

This diagram shows the overall structure of our system, including all of the hardware components and their connections to the ground control station.



Appendix C – Model-View-Controller Program Architecture

The model, view, controller design is extremely powerful because both the model (back-end) and view (front-end) are completely oblivious of each other. As a result, changes can be easily made to either one without updating the other.