# Capstone II - CSCI 4962

## Spacecraft Attitude and Location Detection

Jackson Wark
Mark Gerken
Rohith Perla

Client: Keith Bennett
Faculty Sponsor: Dr. Erin Chambers

3 May 2019
Deliverable #4

# 1 Problem Statement

The objective of the project is to research alternatives to spacecraft attitude detection and position identification. The spacecraft is a CubeSat, a satellite in the shape of a rectangular prism with barebones functionality. It has two cameras on either end, solar panels covering the body of the satellite to provide a limited amount of power, magnetic rods for stabilizing the satellite within Earth's orbit, and a Raspberry Pi Zero single-board computer.

The client, Mr. Keith Bennett, seeks to explore alternative methods of attitude and position detection. Utilizing various inputs in terms of two oppositionally faced cameras on the spacecraft, solar vectors, and time, this project will attempt to produce an alternative method of position identification utilizing machine learning techniques. A proof of concept of a new technique may allow for further development to produce cheaper, lighter, and/or less energy intensive methods of positive attitude detection.

# 2 Background

## 2.1 Star Tracking

The current commonly used system of attitude detection involves "star tracking" to determine the attitude and position of the spacecraft relative to some reference frame based on what stars are visible to the cameras. Star tracking uses an image provided by the onboard cameras. The stars' apparent position in the reference frame of the spacecraft is measured from this image. The stars are then identified so that their position can be compared with their known absolute position from a star catalog.

## 2.2 CubeSat

CubeSats are part of an international initiative started by the California Polytechnic State University, San Luis Obispo and Stanford University's Space Systems Development Lab in 1999. The initiative was designed to facilitate and encourage access to space to students by providing affordable and frequent launches [1].

Students and the general public can take advantage of these launches as a chance to put small satellites in orbit running whatever experiment they would like. A standardized 1U CubeSat is a $10cm^3$ cube that must contain all necessary components internally and adhere to strict specifications set by the CubeSat Program [2].

## 2.3 Raspberry Pi Zero W

The Raspberry Pi Zero (RPi) is a single-board compact circuit board with the full processing power of a fully functional computer. While the zero is only the size of about two quarters, the RPi possesses the capability to run a full Linux-based operating system, provide visual output, interface through USB connections, connect wirelessly to internet connections, read and write to MicroSD, and even connect to Bluetooth devices. All these capabilities are driven by the 512 Mb of onboard RAM and 1GHz, single-core processor [3].

## 2.4 Flight Software / Ares Framework

The Artificial Reasoning for Exploration and Space (ARES) program is a set of software frameworks designed to support the use of simple machine learning, image processing, rule-based processing, and other computing methods in understanding the environment of small exploration systems such as spacecraft, ground robotics, UAVs, or ocean explorers [4]. It is composed of three packages: the ARES Processing Subsystem (APS), the ARES Mission Software (AMS), and the ARES Communication Subsystem (ACS). The AMS provides a framework for flight software while the ACS provides support for a common file system and communication between the other two systems. The APS is essentially the payload of the craft it is deployed on, and provides coordination with a ground station and the execution of 'reasoners'. These 'reasoners' are software experiments like ours, and there will likely be 3-5 of these expected to be deployed in our satellite.

The APS, ACS, and AMS are all 'apps' that are running on the ground and in the spacecraft as well as several other 'apps' within the same system. All of the reasoners are contained within the APS app and share resources. Whenever power is available the system begins a session that executes a 'step loop'; this gathers sensor data, runs the reasoners once, saves a state, and executes again if there is time. The system is designed to run as long as it can and be able to recover from sudden loss of power by saving a state after each step.

## 2.5 Convolutional Neural Network

An artificial neural network is an information processing paradigm that functions somewhat like the biological neural networks that make up biological brains. It is a framework for many different machine learning algorithms (neurons in this metaphor) to work together, supply inputs to each other, and process complex data inputs. A neural network is configured for a specific task, and it learns to perform a task by considering examples of it. With each example, the accuracy of the network ideally improves.

An artificial neural network has a layer of neurons that take inputs, a “hidden” layer(s) of neurons that process those inputs, and a layer of neurons that output a solution. The implementation of a convolutional neural network (CNN) is a particular class of neural network commonly applied to the analysis of visual imagery. A CNN is free from hard-coded filters providing it freedom from the requirement to hand-engineered classification filters. From an high-level overview, a CNN usually consists of convolutional, pooling, dense, and normalization layers. Through these layers, an image (with relatively little pre-processing) can be analyzed segment by segment while the network uses backpropagation to learn different features in the image. Through this process, the network gains the ability to perform complex classification of images.

# 3 Assumptions

## 3.1 Photo Quality

For this project, the assumption must be made that the camera will work as expected and produce pictures of a quality comparable to the pictures used in the device’s training environment. Phenomenon associated with the operation and use of electronics, particularly cameras, in space are assumed affect the equipment to a manageable extent.

## 3.2 Up-to-Date / Current Data

This project will assume that every time the program runs, the program will be able to acquire photos taken with some degree of recency. Timing will exist as a crucial aspect of the calculations, so it will be critical that the photographs / data used for computation be as current as possible.

# 4 Techniques and Tools

## 4.1 Python 3.6

Python 3.6 must be used for this project to interface with the provided Ares framework. Python is an interpreted high-level language that has placed an emphasis on readability. The language is open-source and supports a huge standard library. Additionally, several interpreters (CPython, PyPy, etc.) have been created for the language which can provide different benefits such as increased performance and runtime.

## 4.2 Unity / C#

The simulated environment that will be utilized to create the training data will be created within Unity. This game-engine provides both 2D and 3D capabilities to generate both fantasy-like and realistic environments. Unity provides many tools to developers regarding texturing, advanced lighting, and even a scripting API using the C# language. While other languages have been supported in the past, C# exists as the only non-deprecated language currently supported by the latest version of Unity. As such, C# will play a crucial role in the development of the simulated environment.

C# is a general-purpose, Microsoft developed language supporting many different types of programming paradigms. However, the language was developed with the goal of being a simple, object-oriented language with an emphasis placed on portability [5].

## 4.3 TensorFlow

TensorFlow is an open-source library providing a vast library of tools for high performance numerical computation [6]. This library provides strong support for the implementation of machine learning techniques, and maintains a large collection of highly efficient data structures well-equipped to handle large and complex data.

A relatively recent development is that of TensorFlow Lite. It is a solution for mobile and embedded platforms that promises little accuracy loss, reduced latency, reduced model size, reduced overhead, and optimizations. It also has tools for benchmarking and conversion from regular TensorFlow models to TensorFlow Lite models [7].

## 4.4 Orekit

Orekit is a free, open source, low-level space dynamics Java library that aims at providing accurate and efficient low level components for the development of flight dynamics applications. It is designed to be easily used in very different contexts, from quick studies up to critical applications. It provides basic elements (orbits, dates, attitude, frames, etc.) and various algorithms to handle them.

## 4.5 ECEF Frame

The ECEF frame is a coordinate system that represents positions as X, Y , and Z coordinates and denotes the point (0, 0, 0) as the center of the Earth. All celestial bodies and other objects in the ECEF frame orbit around the Earth. The Earth does not rotate in this coordinate frame.

# 5 Problem Solution

## 5.1 Initial Project Goals

After initial planning and research, the group documented several milestones and goals for this project for the remainder of the academic year. By deliverable two, the group set the goal to have ARES installed onto the Raspberry Pi Zero to begin creating the restrictive operating environment the network would have to perform within. Secondly, there was the goal of having an operational model of a CNN, or a less computationally-intensive model if found to be necessary, able to train, evaluate, and predict some sort of classification for a set of images. The goal for the Unity environment was to have a beta version created with important aspects of the simulation in place, and able to produce images from the simulated CubeSat's position. Finally, the group set to have established some form of data processing that could convert raw images into usable TensorFlow data.

By deliverable three, the group aimed to have all major Unity development complete with only minor tweaks implemented as necessary. During this time frame, large amounts of training data would be produced using various lighting conditions, weather, and the various positions of celestial bodies. Additionally, the group strove to generate some arbitrarily sized dataset, train the model on that data, and establish an accuracy baseline.

Finally, the group's final goals were to deliver the Unity environment with a large and labeled dataset. The CNN would be trained and able to make predictions, ideally with a significantly higher accuracy than the baseline established in deliverable three, and it would be shown the model could successfully run within the ARES framework on the RPi hardware.

## 5.2 Timeline of Progress

Initially, once the goals were set, the group had largely broke into three major efforts. Jackson Wark took the lead on all work associated with ARES and the Raspberry Pi. We determined his effort would focus on adapting the network to work on the RPi and take charge of the installation and running of ARES onto the hardware. Mark Gerken planned to take the Unity work and would be responsible for the development of the simulation to generate the training data. Finally, Rohith Perla initially set to take charge of building the neural network and training and improving it on the provided data. However, very shortly into deliverable two, we recognized hardware limitations of Gerken's personal machine and roles were flipped. After a hard drive failure (likely as a result of intense hard disk thrashing since the onboard RAM was insufficient to support the Unity environment), Gerken and Perla switched roles so that Perla would take the Unity role and Gerken assumed the TensorFlow responsibility.

By deliverable two, we met nearly every goal set. We had a working convolutional neural network (CNN) able to classify handwritten digits from the MNIST dataset. After much struggle with compatibility and package dependencies, ARES was successfully installed on the RPi. The CNN was imported over to the Pi (but not into ARES), and we attempted to run the program on the limited hardware. To no surprise, the training of the network quickly overwhelmed the microcomputer and crashed. It was this point we realized it would be necessary to find out how to pre-train the network, and port over the trained CNN for predictions. The Unity environment had the major components put into place with a high-fidelity model of the Earth, a geocentric orbiting camera, an orbiting directional light representing the Sun, and a sphere object representing the moon. These celestial bodies were placed to allow them to orbit in realistic paths, but not necessarily accurate timings. A script then allowed for the main camera to automatically capture and save its view as PNG files. However, these images were still only just pictures. The group had not met the goal of determining a way to effectively convert them into vectorized data for TensorFlow.

For deliverable three, the most urgent matter was to find a way to vectorize the images. For the Unity environment, we needed a way to produce images with pre-generated labels to use for classification training. We also began the work to find out how to run a trained CNN model on the raspberry pi. For Unity, we found the best way to generate the labels was to place each image in an appropriately named directory. As a proof of concept, Perla used a series of reference points within the simulation to determine what the camera could see. The environment was adapted to determine if the viewpoint of the camera was viewing strictly Earth, space, or a combination of the two (AKA limb). These pictures were then dropped into a similarly named directory. Orbital paths of celestial bodies were initially calculated from ephemerides from the NASA HORIZON web utility. Position and velocity vectors provided exact location and velocity at two week intervals, with all intermediary positions calculated through interpolation to generate estimated orbital paths.

Gerken implemented a simple to use preprocessing system that could take any directory with subdirectories determining the label for all images contained within. The CNN was trained on the initial dataset of 9,856 images, using 80% as training and 20% for testing. From this first training session, we established a baseline of just over 60% accuracy. However, it was assumed that this poor accuracy was a result of an imperfect system of Unity's classification (e.g. some space pictures contained small parts of Earth, not all Earth pictures were completely Earth, etc.) and the intense reduction of resolution causing significant loss of information. For running the network on the Raspberry Pi, we found we could port over the automatically saved checkpoints TensorFlow generates when training the model. By writing a similarly defined model, and copying the checkpoint files, the RPi was found to be capable to make predictions with the same model within its hardware limitations. Overall, the group did stick to schedule and mostly

complete every initial goal set for deliverable three. We succeeded in generating an initial dataset, training on that dataset, and establishing an accuracy baseline. However, the Unity environment did not have as much environmental variability as we were initially hoping with regards to lighting, weather, etc.

## 5.3 Goals for Final Deliverable

The group strove to have the Unity environment capable of outputting paired pictures to directories based off of what major region of the Earth (continents and oceans) the satellite was orbiting above. The pairs were planned to be oppositionally faced to simulate the data expected to be received from the CubeSat in space. The Unity environment was to use parsed ephemeris data to simulate celestial movement. In order to make the process of generating additional training data from the environment easier, there were plans to create a configuration file allowing for parameterizable generation of data.

The Convolutional Neural Network (CNN) was to be adapted from taking just one image as input, to taking in a pair of images like the satellite was supposed to provide. It also was to increase the input image size from 28x28 to a size that would involve less interpolation of the original larger images and hopefully increase prediction accuracy. The network was also to be modified to consider images as sequences and to give consideration to previous classifications as information for future guesses. Initially, the goal for deliverable four was to deliver a model that outperforms the baseline. However, the model ended up being modified more after deliverable three to allow for more specific classification, which would expectedly hurt accuracy scores.

Since the network was successfully exported and run on the RPi in the previous deliverable, the next issue was that of performance. First, the memory size and runtime was to be examined and quantified to gain a better understanding of resource usage. It was also decided that the relatively new TensorFlow Lite toolset would be tested for feasibility and performance compared to standard TensorFlow. One major goal that was not yet achieved was actual integration of the neural network into the ARES framework, so it had to be satisfied for the final deliverable.

## 5.4 Final Results

The camera in the Unity environment is configured to orbit around the Earth, spin while orbiting at a parameterizable rate, and output pictures to directories based off of what major region of the Earth it is above. There are eleven major regions that are checked for in the simulation: the 7 continents, and the Pacific, Atlantic, Indian, and Arctic oceans. This was done by using Unity's `Vector3.Angle()` and `Vector3.SignedAngle()` functions to find the angles between the satellite and Unity's up and right vectors ((0, 1, 0), and (1, 0, 0)) respectively. These angles effectively recreate latitude and longitude coordinates. Then, on a world map, coordinate ranges

were devised for all the major parts of each region (e.g. for North America, a coordinate range was created for the USA along with coordinates for Canada, Alaska, Mexico, Central America, and so on). Afterwards, in Unity, those coordinate ranges were listed, and if the latitude and longitude fell inside those ranges, then the camera would consider itself above the region that range belonged to.

The goal to generate pairs of oppositionally faced images was removed in mid-March. Mr. Keith Bennett informed us that, due to a hardware problem, the space lab team had opted to include only one webcam onto the CubeSat. As such, this functionality was no longer necessary.

The Sun and Moon in the Unity environment are configured to orbit around the Earth based off of the coordinates in two data tables, one for each celestial body. The data tables contain the locations of the Sun and Moon according to the ECEF frame at every hour over the course of two years, starting on January 1st 2019 at 0000 UTC and ending on December 31st 2020 at 2300 UTC. To obtain the tables, Orekit was used to lookup the coordinates for the Sun and Moon and save them to CSV (comma-separated values) files. In Unity, the CSV files are read and the simulation interpolates between the coordinates specified in the files. The simulation is also configurable to start at any date and hour in those two years and proceed from there.

The CNN is a five layer network consisting of a batch of processed images being sequenced through a convolutional and pooling layer, a secondary convolutional layer and pooling, a dense layer of perceptrons, and then outputted into a prediction vector. Originally, the CNN reduced the received images into a 28x28 vector. This caused drastic interpolation of features and resulted in an accuracy barely over 60%. Shortly after deliverable three, Gerken figured out how to adjust the layers to allow for a higher-featured vector (up to 112x112) which immediately increased the accuracy to over 80%.  Overall, the focus on how to improve the CNN shifted quite drastically from the original goals. The goal to increase the resolution of the input was completed as planned; however, the functionality to interpret pairs of images was removed completely due to the removal of the second camera. Research was done on sequencing the data and considering previous classifications to aid future predictions, but ultimately implementing this proved to be quite difficult and Gerken found little evidence that doing so would benefit the classification significantly. Finally, the goal to improve upon the baseline was reached very early on regarding the ELS classification. However, accuracy reduced significantly, as expected, upon the implementation of the major bodies classification.

Throughout the majority of the deliverable, the effort to improve the model focused on identifying optimal hyperparameters. After researching different tuning methods, Gerken initially attempted to implement Bayesian Optimization. After basic research, and a few failed implementation attempts, it was decided it would be best to pursue a simpler optimization

strategy. A grid search strategy would have been quite resource intensive and slow. Instead, a random search was implemented after research showed that random searches often yielded a very similar tuning configuration, but at the cost of the losing the guarantee of optimality. Since then, the CNN has been modified to allow for the vast majority of the hyperparameters to be automatically selected (randomly from a predetermined range) over a specified number of trials. A defined dictionary outlines a key for each of the hyperparameters in the model. Each key is associated with an array or range of possible values. At run time, a trial is initialized by creating a new directory to maintain the TensorFlow checkpoints, and random values are selected from the dictionary to initialize the value of the hyperparameters. The model is trained on the dataset, evaluated, and the hyperparameter values, accuracy, and runtime is recorded. Once the trial is complete, the values are randomly reinitialized, a new directory for checkpoints is created, and the process repeats on the same dataset.

Once this program was created, the program and dataset was ported over to Hopper. At the time of execution, this dataset still consisted of the Earth-Limb-Space (ELS) level of classification. Gerken ran the program for 500 trials with all results being recorded to a CSV. Overall, the total time of execution took over four days to complete. The changing hyperparameters resulted in an accuracy range of 70-85% accuracy and a runtime that varied between 3.1 and 19.7 minutes. However, it's important to note two potential problems readily apparent with this method. First, the runtime is being calculated based on a difference of a real-world time clock rather than a measure of actual processing time. On a Linux cluster like Hopper, external demands placed on the system by other users and processes could make a notable difference in the real-world runtime of a program even under identical contexts. Secondly, running and testing the same dataset over many trials runs a strong risk of overfitting the data. A potential solution to both of these problems are discussed in section 5.5.

Finally, once the Unity environment was capable of creating the more specific dataset identifying the major bodies of Earth, the neural network output was adapted to produce a prediction vector for the major bodies. The breakdown of the dataset is as follows:

```
Total:          87,804
Africa:          8,805 (10.0%)
Antarctica:          0
Arctic:              0
Asia:           11,208 (12.8%)
Atlantic:       17,641 (20.0%)
Australia:       2,451 ( 2.8%)
Europe:              0
Indian:         10,513 (12.0%)
```

```
North America:    6,819 ( 7.8%)
Pacific:         29,767 (33.9%)
South America:    2,579 ( 2.9%)
```

While 11 different bodies exist, it can be seen that the orbital path of the satellite does not pass over all the regions. As such, several of the regions currently have zero data to support their classification. Due to time constraints, Gerken only ran a single trial of the model on Hopper which produced the following results:

```
Trial 000
kernel_initializer, <class 'tensorflow.python.ops.init_ops.Ones'>
conv1_filters, 108
conv2_kernel, 4
learning_rate, 0.019000000000000003
conv1_kernel, 3
conv2_filters, 76
units, 512
dropout, 0.30000000000000004
activation, <function leaky_relu at 0x7f2c2d81a050>
--------
loss, 1.4424771
global_step, 1098
accuracy, 0.44711512
run_time, 5782.31662393
```

As can be seen from the results, the model yielded a 44.7% accuracy. Additionally, due to the size of the dataset, it took over 96 minutes to complete one trial of training and evaluating the neural network. This demonstrates a proof of concept that the network can achieve some degree of precision in identifying its position, but requires significant more work before it could be implemented as a reliable system.

Tests were undertaken to determine the impact the CNN and ARES would have on the RPi, and how much more memory there was to work with. Even though the RPi has the already limited 512 MB of physical memory, not all of it is available for general use. Some is owned by caches and buffers, so when the linux command `free` is run it shows that there is about 295 MB available for our use. Using the linux `time` utility peak memory usage statistics were recorded for the CNN in it's Earth-Limb-Space (ELS) classifier form as well as running the user interface that runs the reasoner in ARES.

The results follow (where memtest is a bash alias for the time utility):

```
memtest cnn_classifier
max memory size 147400K
elapsed time: 70.97

memtest APSUI.py
max memory size: 118040K
elapsed time: 42.87
```

It is clear to see here that the CNN used about one half of the available memory resources of the RPi, while ARES running for one step appears to use about 40%. These programs are meant to be run at the same time, and if simply summed they still would not quite use 100% of the RPi's RAM. That said, it is unlikely that if both programs were integrated they would use 265 MB, as this number is peak usage and likely incorporates some overhead as well.

In order to implement and test TensorFlow Lite and an alternative method for exporting the CNN to the RPi, the latest TensorFlow version, 1.13.1 was installed on the test environment as well as the RPi. Because TensorFlow Lite is so new and still in rapid development, using 1.13.1 would be necessary to take advantage of the best possible compatibility, support, and tools. Despite compatibility issues in the past with the TensorFlow install, 1.13.1 was successfully installed, with only one problem and workaround identified involving a compiler error and the `h5py` package.

In the end no working solution was found using TensorFlow Lite, though TensorFlow 1.13.1 was found to be compatible with the RPi. Instead the checkpoint method from Deliverable three is the final method being used at the present time. In this method an automatically saved `.ckpt` file that stores data on the CNN's trained weights and variables is used to start up that network on a given device. This only works if the graph of the network is defined exactly the same in the file running the restored network as it was defined when the network was trained. A `tf.Estimator` object created in the same way as for training can restore the network and run it's `predict()` method to generate predictions on data that is prepared in the same way it was processed for the network during training.

Using the checkpoint method, the CNN was integrated into the `APSReasonerLOCORI.py` file. The file is a class definition defining a child of Reasoner in the ARES APS subsystem. The CNN was easily integrated into this file by taking the model definition code from the training script that defines the network's graph, and putting it inside a function called `cnn_model_fn`. Another function called `CNN` was created that creates the `tf.Estimator` object, prepares the data for

prediction from the directory `/tmp/predictions`, and runs the `predict()` method on this newly prepared data. Then a call to tf.app.run() which runs the specified TensorFlow program was added the step method of the `APSReasonerLOCORI` class and targeted the `CNN` function. Running a step of the reasoner then calls a single prediction from the CNN on the available data in the `tmp/predictions` directory and outputs it's result to the terminal.

## 5.5 Unfinished Work

For the CNN, there are a large number of changes that would have ideally been completed and questions to be answered had more time been available. First, not all hyperparameters were able to be subjected to the randomized search. Several tests were conducted to determine an appropriate input size that balanced between performance and accuracy. However, these tests were far from comprehensive. Ultimately, it was settled to use a 112x112 vector to hold the inputs but further experimentation might find a more optimal value. Additionally, the convolutional layer filter size is currently fixed at 64 because modification of that hyperparameter it affects the shape of the output layer, causing problems in the reshaping and flattening of data. Next, it would drastically reduce the computation time of running trials in a randomized search if the program was written so as to run multiple trials in parallel. Due to the independent nature of the trials, parallelizing the operation should be a relatively natural conversion. The documentation of runtime should be improved so as to use CPU/GPU time instead of real-world time. This will be especially necessary if parallelization is implemented. To prevent overfitting, Gerken had begun to implement the use of a validation dataset. Instead of training each trial on the same dataset broken up into 80% training and 20% evaluation data, it would have been more beneficial to have created a validation set as well. Ideally, the program would train on 70% of the data, evaluate on 20% of the validation set, and record those results. This process would repeat using these two sets for the remainder of the trials. Once trials were completed, review of the results would identify the higher performing models. The selected models should then be evaluated again on the remaining 10% of data that they had never seen before. This would help determine if the higher performing model truly was performing better or if the tweaked hyperparameters overfitted the model to the data. However, this became difficult to implement as it required a way to save and reuse datasets. Efforts had been made to save and serialize the data to allow for reuse in the past, but failed due to an inability to pickle the data. This failure is a result of a lambda function being used in the building of the dataset which is unable to be pickled.

Perhaps the most obvious shortcoming of a set goal is that of implementing TensorFlow Lite. Though unforeseen at the beginning of the project, it was a goal stated after Deliverable three. An effort was made to implement a solution, but ultimately time constraints and compatibility issues meant it was not able to be deployed at the time of this Deliverable. Since TensorFlow Lite is so new it is constantly being updated, improved, and re-structured. In order to stay current

and utilize new tools and compatibility efforts, an upgrade to the latest TensorFlow 1 version, 1.13.1 seemed necessary. This change was made, But also caused some compatibility issues with our model that was written in version 1.9.0. TensorFlow Lite has a model converter that translates certain files into TFLite compatible models. These include `SavedModels`, frozen `GraphDefs`, and `HDF5` files from the `Keras` API. Since the group was using `tf.Estimator` API the `HDF5` files were eliminated. Since the `SavedModel` format is mainly designed for use in TensorFlow Serving, client-server based prediction, frozen `GraphDefs` seemed the appropriate approach. Using a `GraphDef` file and the desired checkpoint (`.ckpt`) file a frozen `GraphDef` can be created. What this is doing is taking a data and variable-less graph definition (`GraphDef, .pbtxt`) and combining it with all of the data and variables (checkpoint, `.ckpt`). This was done with a provided script `freeze_graph.py` without any significant issues. But when using the TFLite converter on the resulting file, errors were returned. These errors related to the fact that TFLite only supports a limited number of operations that are available in core TensorFlow. So many of the operations in the CNN were either not supported or had equivalents that needed to be specified manually. After trying some official work-arounds it seemed the only options were writing custom operations and custom TensorFlow kernels, or designing a network with the supported operations in mind. There were also numerous errors with an operation called `RandomUniform` that seemed to be a component of many other operations. These appeared to be unsolved issues related to the incomplete nature of the TFLite library. Given more time these issues could be worked around, solved, or taken into consideration and the CNN re-designed.

# 6 References

[1] “The CubeSat Program,” CubeSat.
[2] “CubeSat Design Specification,” The CubeSat Program, CalPoly SLO, Rev. 13. 12 Apr 2014
[3] “Raspberry Pi Zero W,” RaspberryPi.
[4] Bennett, Keith. “ARES APS Software Design Document,” Bennett Research Technologies, Dec 2018
[5] “C# Language Specification,” Ecma International. June 2006.
[6] “About TensorFlow,” TensorFlow, Google Brain Team.
[7] “TensorFlow Lite,” TensorFlow, Google Brain Team.
[8] “Introducing the Model Optimization Toolkit for TensorFlow,” TensorFlow, Sep 18, 2018