Runway Location for Autonomous Aircraft Using Neural Networks and Computer Vision

A Thesis

Presented to the

Department of Computer Science

and the

Faculty of the Graduate College

University of Nebraska

In Partial Fulfillment
of Requirements for the Degree

Master's of Science in Computer Science

University of Nebraska at Omaha

by

John Ludeman

July 2022

Supervisory Committee:

Dr. Qiuming Zhu

Dr. Xin Zhong

Dr. Chenyu Huang

Abstract

Runway Location for Autonomous Aircraft Using Neural Networks and Computer Vision

John Ludeman, M.S.

University of Nebraska, 2022

Advisor: Dr. Qiuming Zhu

Recent advancements in neural network architecture coupled with increased graphics card processing power have allowed for the rise of real-time prediction. Proposed is a method of using a semantic segmentation neural network followed by computer vision methods and lens equations to recover the location of a runway and the altitude of a drone from an image and a known runway dimension. The best performing neural network model had an intersect over union accuracy of 95.5% with an inference speed of 12.5 frames per second. The final error in predicted altitude using the best of two developed methods was less than 15% on average, with a peak error of 23.5%.

Table of Contents

Lists of Multimedia Objects

1. Introduction

1.1. Problem

Autonomous aircraft are becoming more and more common as control and locational technology advances. However, many systems are still reliant on ground stations, human input for landing, or dumb landing. For example, one of the most common consumer autopilots, ArduPilot [Ardu Dev Team, 2021] is only able to land at a waypoint. This waypoint is either user-created or is the takeoff location without any capabilities of the craft to identify suitable areas. This landing does not take into consideration the conditions of the area at the landing waypoint.

Modern neural networks have been able to identify and segment regions of interest to high levels of accuracy. A network that can determine the region of an image where a runway is located would provide a new source of information allowing for the correction or overruling of erroneous or lack of data.

1.2. Motivations

Current landing systems require precise information to land. Some airplanes have the ability to autoland at approved airports through a combination of many sensors, including a radio altimeter and the Instrument Landing System (ILS). The ILS system is composed of two radio beams that provide precise vertical and horizontal guidance to the plane through a radio localizer.

Another source of information for landing is the Global Positioning System, a system of satellites, upkept by the US government, which can triangulate the position of a receiver. However, according to the US Government, GPS error is <= 2m with a 95% probability [“GPS Accuracy”, 2022]. While the error is not particularly large, the possibility of error without a method of determining if there is error can lead to serious consequences. GPS error is also not static, meaning the error cannot be corrected a single time as the error will drift. In drones, the error in GPS is no less detrimental. A more accurate source of information for landing in commercial planes is the radio altimeter, which has an accuracy of ± 0.75 meters [Labun et. al. 2019].

Using image segmentation or bounding boxes to find the regions of interest and applying computer vision techniques, the dimensions and location of the runway can be determined. If the location and dimensions of the runway can be calculated from an image and are known to the drone, the exact position of the drone relative to the runway is determinable. This can lead to landings being performed possibly without radio guidance and without ground station assistance.

1.3. Significance

The successful application of the method outlined in this thesis would allow for an autopilot system of an autonomous aircraft to land at a previously un-cataloged runway without infrastructure for landing. Beyond hobby uses, a program that can provide runway locations to the autopilot would benefit commercial ventures. Many delivery companies are exploring autonomous delivery using drones.

Amazon's Wing Aviation LLC has developed a combination of a fixed wing drone with vertical component, shown in **Figure 1.**

Figure 1: Wing Aviation transport drone. Image from https://wing.com/how-it-works/

Fixed wing aircraft require more space for landing, with a higher landing speed when compared to vertical take-off and landing capable craft. A quadcopter does not need to know the exact altitude above the terrain to land, as it can slowly descend at a landing speed from a reasonable height to mitigate the effect of error. Fixed wing aircraft do not have this luxury because if they do not touch down quick enough, the plane overshoots the runway. The speed of the craft is also an issue, as the minimum landing speed will still potentially lead to more damage in the event of an accident. Though the landing and takeoff profile is worse for fixed wing planes compared to vertical takeoff and landing (VTOL) capable craft, the craft can carry significantly more weight, more efficiently.

If fixed-wing cargo drones can detect a runway accurately and determine where it is relative to the plane, less infrastructure may be required for landing. Ground bases are not required to provide radio guidance, the runways can be shorter since the exact location of the craft and runway are determined, and GPS error can be constantly corrected for. This could lead to light aircraft being used to quickly transport goods from location to location.

1.4. Challenges

Outlined below in section **2.2 State of The Art** are three the previous implementations of runway recognition and runway locating. The previous methods researched are not available to the public, not able to locate the runway, or cannot operate in real-time on a computer, let alone an embedded system.

The proposed method also uses the pixel distances and a simplification of lens equations to calculate the distance and location of the runway. As the camera quality decreases, so too does the accuracy and reliability of a system dependent on the camera. The propagation of error can cause issues in the final predictions, so a method of disregarding bad predictions is needed.

1.5. Objectives

The program developed in this research can do two distinct things: discriminate and generate runway information for autonomous systems. With video, GPS, and pitch

information, the program can discriminate images where there is no runway and images where a runway is in the frame within a margin of error. The program is also able to generate some of the needed information for a fixed wing plane to land, namely the location of the landing area relative to the craft and the altitude of the craft over the landing area.

The program was initially planned with the idea of near real-time prediction but that was not completely achieved. The final program runs at an average of 12.4 frames per second, with slowdowns depending on the quality of the segmentation and the size of the runway in the image.

2. Overview

2.1. History of the problem

Autonomous landing of drones has been an issue since the development of autopilot. When in flight, an error of a few feet due to barometer and GPS error is not an issue. If the craft is at 450ft when reporting 500ft above ground level, a catastrophe is not imminent. Commercial barometric altimeters can have an error of up to 75 feet, while still within an acceptable margin [Federal Aviation Administration, as of 2022] The landing of hobby drones has had a similar ethos if a bit laxer. In the autopilot mentioned above, ArduPilot, the landing is not accurate, only landing at a user-designated location (take off or waypoint). When a model plane flairs for landing using ArduPilot, it bleeds off enough speed and energy that an error of a few feet still does not damage the plane if it lands on a soft surface.

Another reason that this has not become an avenue for research until recently, is due to the advancements in machine learning and neural networks. Neural networks have seen a significant increase in capabilities and applications in the past decade. Breakthroughs in model architecture have increased the processing speed of neural networks. Graphics cards are more powerful with each successive year, reducing processing time even more. A neural network that can now run in tens of milliseconds may have taken tens of seconds on a system from as little as half a decade ago.

Many diverse groups have worked on methods of runway detection, including large aircraft manufacturers like Airbus, though very little of the commercial research is available to the public.

2.2. State of the art

Airbus has developed and tested their Autonomous Taxi, Takeoff, and Landing (ATTOL) project as a solution to the problem researched in this thesis. The first testing phase was completed successfully in 2020 [Duvelleroy and Benquet, 2020]. With no response from Airbus to communication, the following is mostly speculation on how ATTOL works. 450 test flights were conducted to gather video data, which posits that the program uses a neural network due to the total amount of information gathered. The image below shows ATTOL specifically following white runway lines, instead of the total runway edge, so computer vision techniques such as Hough Transforms may have been used. Since a single image is shown in the article, it may have been the best-case scenario of the program and does not give any great hints as to the actual method. Other airlines are likely working on this problem, but no others have public information available.

Figure 2: Image provided in Airbus ATTOL [Duvelleroy and Benquet, 2020]

The method explored in “A Robust Vision-based Runway Detection and Tracking Algorithm for Automatic UAV Landing” [Jbara et. al., 2015] attempts to create a model to match the runway. The model, shown in **Figure 3** describes the runway shape based on a semi-static model generated from a set of tests conducted by the Saudi Aerospace Research Center. The four distances try to generate line segments that separate the most differing mean RGB values. The algorithm then tries to minimize an energy defined by these separations.

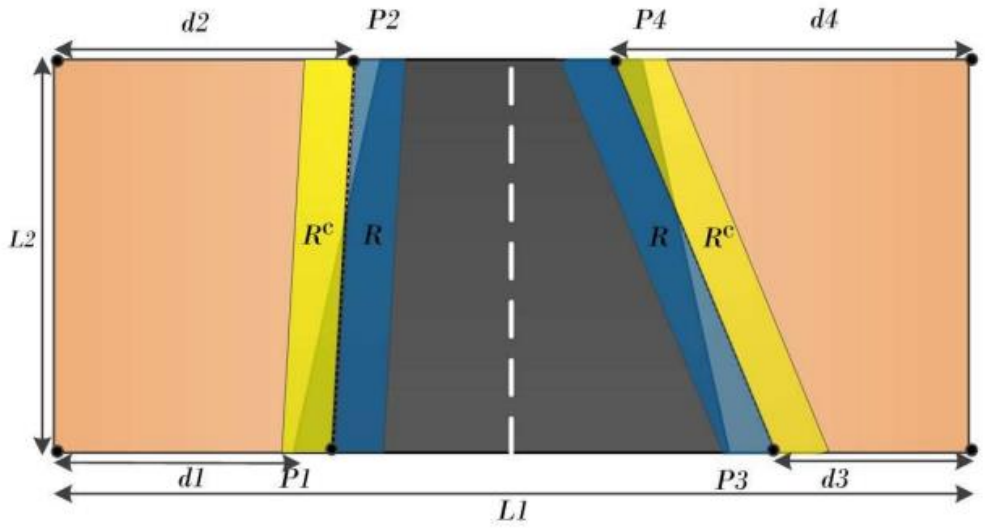

Figure 3: Model to match runways [Jbara et. al., 2015]

This method relies on a set of assumptions: the drone must be within 10° of the runway heading and have GPS accuracy of less than 15m. The second assumption is quite strong, as GPS is guaranteed to be accurate within a few feet 95% of the time. To be within 10° of the runway heading means the drone already knows the location and orientation of the runway, which is part of the solution developed in this thesis. As shown in **Figure 3**, it is assumed that the craft is above and in line with the runway. This only helps in the cases where the craft knows the exact location and orientation of the runway already. This

method only provides the precision needed for landing with GPS, where the runway is already known to be in frame, rather than determining the location of the runway.

Semantic segmentation is the separation of a region of interest from the rest of an image. An implementation of region segmentation like that in this thesis was done by Mosaic Data Science [“Detecting Airport Layouts with Computer Vision,” 2021]. Mosaic Data Science used a convolutional neural network called U-Net to segment satellite images of runways. **Figure 4** is from a presentation by Mosaic Data Science on their software. It shows the mask (in red) overlayed on the original image shown in **Figure 4**. The accuracy is quite remarkable, as the program can discern runway from taxiway.

Figure 4: U-Net applied to satellite images of runways [“Detecting Airport Layouts with Computer Vision” 2021]

U-Net was initially developed for the segmentation and tracking of cells for biomedical purposes [Ronneberger et. al., 2015]. One of the main goals of this CNN was to reduce the total amount of training images needed as the initial target segmentations were extraordinarily complex and limited. U-Net trains and predicts using a mask, which is a binary (or more) overlay of the input image. This overlay or mask is the individual

regions to be segmented in the image with the labels. If U-Net is trained on binary masks, the segmentation is simply separating a region of interest from the background. With more values in the mask, the regions of interest can also be classified.

The benefits of using U-Net are the accuracy of the segmentation and the low amount of annotated training images. These benefits come with a steep price, however. U-Net is formed of many different convolutional layers, shown in **Figure 5,** leading to a long prediction time per image. In the original implementation of bio-medical images, the prediction time was one second. While the article authors considered this to be a great time for their purposes, this speed is too low for near real-time prediction. The prediction may be sped up by reducing input image and mask size, reducing the total layers, or decreasing the parameters in the layers. The experiments explained later in **Chapter 5 U-Net Models for Runway Location** attempted to reduce computation time and increase accuracy.

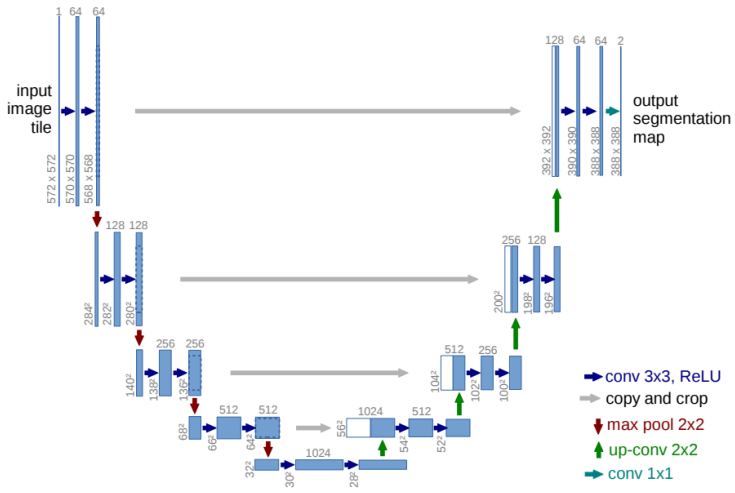

Figure 5: Architecture of U-Net [Ronneberger et. al. 2015]

3. Images and Target Generation

3.1 Source Images

The base images used for training and validation in this research were provided by Sawyer Buller, a student at the Aviation Institute of the University of Nebraska Omaha. The pictures were taken from video captured using a DJI Phantom 4 Pro, at a resolution of 1080x1920p at 30 frames per second. The drone was flying slowly around the runway at many different altitudes and distances. The labeled images were taken once every second due to the similarity of temporally close frames. The more exact data points for testing the image algorithms as outlined in **Chapter 8 Transformations from Pixels to Real Worl Locations**, were from this same runway but 3 months later after all the flora had greened.

Figure 6: Images from the training dataset. Center 1080x1080 of original image resized to 512x512

These images would be too large for most modern desktop computers to train neural networks on, so the images were truncated to the center 1080x1080, and then downsized to 512x512 using the bilinear interpolation method provided by the Python package OpenCV. **Figure 6** shows three images from the downscaled testing data set.

The primary recording location was Hawk Field, home of the Omahawks Radio Club. The club website is https://www.omahawks.org/. Hawk Field is a well-kept model runway with a mock taxiway and painted lines. This runway is the only one in a large area around Omaha that allows for the flying of drones. Most other runways are on city park land, which has disallowed the use of drones. For this reason, only high-quality footage is available from this location with the other runway images being taken with a phone. The lack of photos of other runways will hamper part of the initial proposal of evaluating the accuracy on other runways and the training time to a new accurate model. The accuracy will still be assessed in this research, but the video and photos will be extremely limited in scope.

3.2 Target Generation User Interface

The user interface (UI) for generating the targets used in training and testing is shown in **Figure 7.** Four exclusive buttons for selecting the four corners of the runway with buttons for moving the corners in different increments. By toggling the boundaries, the corners can be outside of the image for when the runway is only partially visible. The goal is for the lines to follow the runway lines, not necessarily that the four corners be on the runway corners. When the runway is only partially in the image, a polygon may need five or more corners to represent the shape if the corners are limited to the image dimensions. In the image shown in **Figure 7** for example, the minimum number of corners required to circumscribe the runway is six if the corners were limited to the image. However, if the bottom two corners were moved outside the image so the

connecting lines follow the runway edges then there is no issue as the entire runway in the image would be encompassed by four corners.

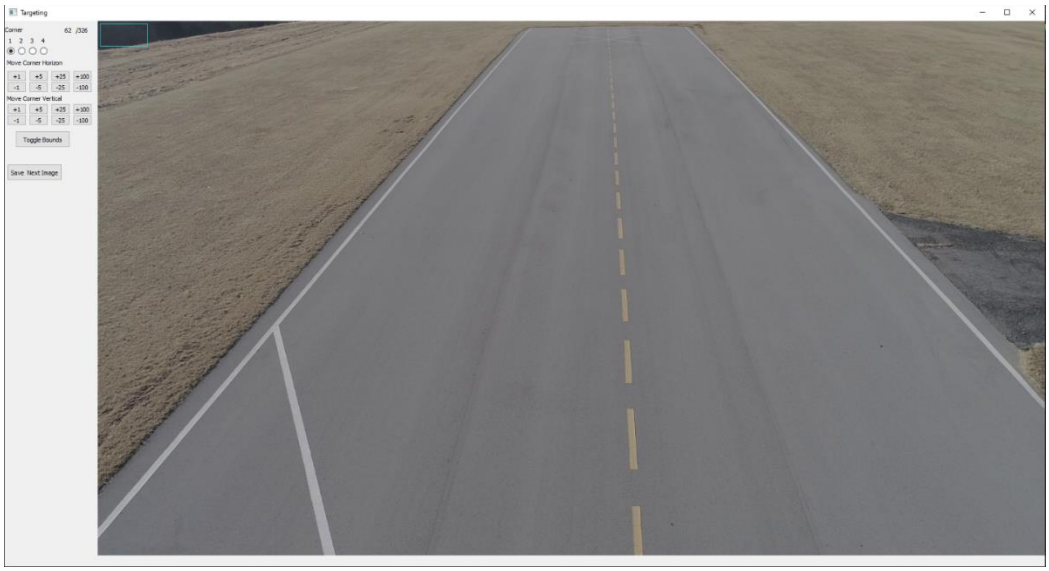

Figure 7: Target UI

The corners are moved using buttons to increase or decrease the row/column position with the added ability to use the arrow keys for more fine and quick movements. Some error occurred in the labeling of the photo due to the distance to the runway at some points. When the runway takes up a small portion of the image, it is easy to miss the corners. Other issues are fatigue in the human labeling the images. As more images are labeled, the person labeling the images may become lax in the accuracy of the corners. These are not a major issue as the small errors in labeling should be outweighed by the error inherent in any machine learning algorithm.

The corners are saved with the video name and specific frame name with the corner row, column in a CSV for easy parsing later.

3.3 U-NET Specific Targets

Since the U-NET target is the segmentation of the runway, meaning all points of interest are non-zero, the quadrilateral formed by the four corners must be filled in. The generation of this filled in mask is accomplished in two steps: first the four corners must be sorted to be clockwise from the barycenter (average of all points), then a given point must be tested to see if it is within the quadrilateral.

Sorting the corners is accomplished using the cross product between the two points based off the barycenter of the quadrilateral. The barycenter is the average x, and the average y of the four points, creating a center of mass that will always be within the shape. The cross-product equation is:

$$\bar{A}x\bar{B} = \left||A|\right| * \left||B|\right| * \sin(\theta) \qquad \text{[EQ1]}$$

Since the magnitudes of vectors, or in this case a vector to the point, are always positive, the sign of the cross product will indicate the direction of the angle between the two points. If the angle between A and B is negative, then B is clockwise from A. For example, two points on the shape are as shown in **Figure 8** below:

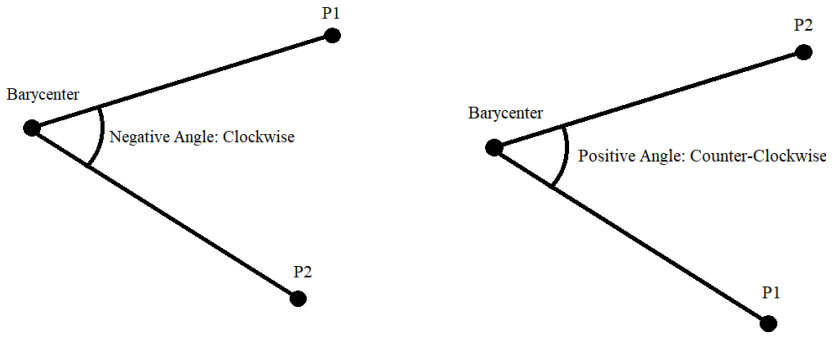

Figure 8: Left: P2 is clockwise of P1. Right: P2 is counterclockwise of P1

In the left image of **Figure 8**, the cross product between the two vectors would generate a negative value while the right image generates a positive value. Positive angles move

counterclockwise when compared to the reference point, and negative angles move clockwise. A simple sort algorithm is used to sort the points based on the inequality determined by the cross product of the two vectors.

With the four corners of the box sorted, a point can be determined to be inside or outside the box using the four triangles generated by the four corners and the point to be tested. If the point is labeled as P and the corners as A/B/C/D clockwise, the four triangles would be ABP, BCP, CDP, DAP. **Figure 9** below shows how this test works.

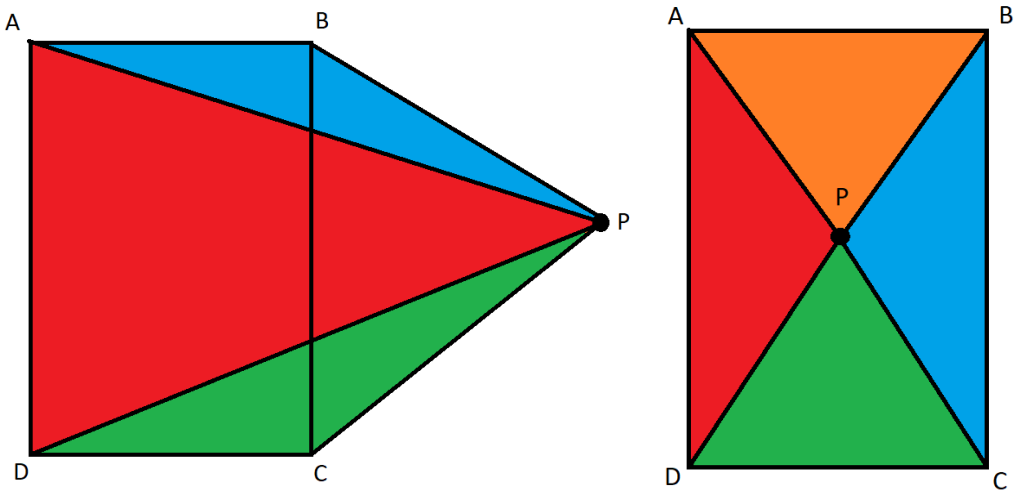

Figure 9: Left: Point P outside of box. Right: point P inside of box

With the point outside of the box to the left, the areas of the four triangles will add up to be more than the area of the quadrilateral. If the point is inside the quadrilateral, the triangle areas will add up to be equal to the quadrilateral area. The user Mili gave this elegant answer [Mili, 2013] on Stack Overflow. The method only works on convex quadrilaterals, i.e., quadrilaterals where the inside angle is less than 180°. Since the

corners are the runway corners or trying to mimic the corners when they are out of frame, the quadrilateral is always convex.

The area of the quadrilateral is calculated using Shoelace theorem, also known as the Surveyors Formula, [Bradon, 1986] which takes a set of clockwise ordered point to determine the area of any polygon. When the shape is a quadrilateral as described above, the Shoelace theorem equation is:

$$\text{Quadrilateral Area: } \frac{1}{2}(Ax * By + Bx * Cy + Cx * Dy + Dx * Ay) \qquad \text{[EQ2]}$$

With the area of the quadrilateral and the area of the four triangles generated from the four corners and the point, a simple equality will determine if the point is inside of the region. If the two areas are equal, then the tested point is in the region.

Finally, the application of all the above will generate the following target image, as shown in **Figure 10**.

Figure 10: Left: Image is the base image. Right: Target overlayed on the base image.

These calculations were done in C integrated into Python. Some issues arose where the input into the C programs from the target generation Python program must be as a vector due to memory access errors, so transformation from 1D coordinates to 2D or 3D coordinates was needed. The speed increase across all images was near a third of a day. Each target to process would take a minute in Python, but less than a tenth of a second in C.

3.5 YOLOv3 Target Generation

The targets for YOLOv3 are the bounding box center X, center Y, width, and height scaled for the image dimensions. Initially, the target generation for YOLOv3 was quite complex. The first version of both the U-Net and YOLO targets used slices of the entire image with overlap to turn a single labeled 1920x1080p image into many. Due to the chopping of the runway in the image slice, the YOLO values needed to change. To calculate the new values, the corners were moved along the connecting lines using the line equations. This method was over one hundred lines when coded for all possible edge cases, such as division by 0.

Once the robust U-Net generation was complete, YOLO values are simply based on the U-Net targets. By looping through the masks for the max and min of the rows and columns, the YOLO target generation is quite simple.

3.4 Issues with Target Generation

There are issues with this method of getting the area filled in, primarily due to the inclusion or exclusion of certain boundaries and corners. The method above does consider the corner points to be inside the quadrilateral, but some of the points along the bounding box are not, due to the transfer of a continuous equation to discrete pixels. If the image from **Figure 10** is enlarged, this issue becomes apparent.

The row difference between the two top corners is a single pixel, so all intermediary points on the line between the two are not a whole number. The true cause of this error is the nature of pixels. A pixel in the topmost row, row zero, will contain all row values [0, 1). By iterating through row and column pixel values, there is no calculation between row values, so the values (0, 1) are lost. Additional calculations could be done to improve the tracking, such as if most of the row values covered would be within the target but the error is not large when compared to the other sources of error. This line exclusion error is shown in **Figure 11**.

Figure 11: Corner inside, rest of line outside on target

Another source of error in the target is caused by mother nature and human error. Since the test runway for this research is only a mockup of a runway and not a regulated airport by the FAA or other agencies, the precision is not great. In this same vein, the edges are also broken due to the grass overgrowing the runway. These two combined cause the

edges to not be true lines. Both human and mother nature error is shown in **Figure 12** below. The right-side white line is as true as possible for hobby plane enthusiasts, but the line converges and diverges from the edge of the concrete. In the lower threshold, the grass has overgrown the edge of the concrete.

Figure 12: Poor concrete edges create poor targets

4. Brief Overview of Convolutional Neural Network Layers

4.1 Convolutional Layer

Convolutional layers were first explored by Kunihiko Fukushima in his paper "Neocognitron: A Self-organizing Neural Network Model for a Mechanism of Pattern Recognition" **[**Fukushima, 1980]. A modern-day convolutional layer looks quite different from the neocognitron, though the basic principles are the same. The idea that localized changes in pixels are one of the main determining factors of classification.

The convolutional operation takes a mask and applies it to the input. For example, if a convolution mask is 3x3, the output will be the elementwise multiplication of the matrix and the pixels, then the sum of the nine values to produce the single output at the same location of the center of the 3x3. Generally, the output channel dimensions are larger than the input. In the U-Net model shown in **Figure 5,** the first layer input is a 572x572x3 and the layer output is 570x570x64**.** This increase in the number of channels is one of the main parameters to be changed in a convolutional neural network. For example, the filter count in this layer (64) is the number of different 3x3 matrices with different weights and biases applied to the image.

The reduction from 572x572 to 570x570 is due to the mask being outside of the image range. If a 3x3 mask is applied to the very top left pixel, then both the top row and left column of the mask are out of bounds. So, the entire top row, bottom row, leftmost column, and rightmost column do not have the mask centered and applied. As more convolutional layers are applied, the output shrinks. In the case of the basic U-Net, the

final output size is 388x388. This was not desirable as the final output target disregards over 50% of the image. To prevent the reduction of output dimensions, all values outside of the image can be considered zero so the convolution mask can still be applied on the edge pixels. This is called image padding.

4.2 Max Pooling Layer

Max pooling is self-explanatory. For a given shape, reduce the pixels in that shape to only the max value found in that window. If a window is 2x2, it will reduce four values to one reducing the shape in both height and width by half. Since the object to be identified can be anywhere in the image, max pooling tries to remove the location of the object in the image.

4.3 Transposed Convolution Layer

Transposed convolution tries to undo the effect of pooling. Max pooling reduces the height and width by a set amount based on the size of the window, transposed convolution learns values for a window that increases the height and width (and perhaps channels).

4.4 Dropout Layer

The dropout layer randomly selects inputs to be zero at a set rate. By randomly removing inputs, dropout attempts to decrease overfitting. Overfitting is the tendency for a model to become too perfect on the training data, that the accuracy falls for data on which it has not trained.

4.5 Dense

The dense layer is a fully connected layer, where every single input element influences the output element. Dense layers are usually used at the end of a convolutional network to generate the class probabilities due to the large parameter count. It is not used in U-Net.

5 U-Net Models for Runway Location

5.1 U-Net Overview

U-Net was created by Olaf Ronneberger, Philipp Fischer, and Thomas Brox and outlined in their paper "U-Net: Convolutional Networks for Biomedical Image Segmentation" [Ronneberger et. al., 2015]. The network is quite aptly named due to the shape of the interconnecting layers of the network. The original architecture is shown below:

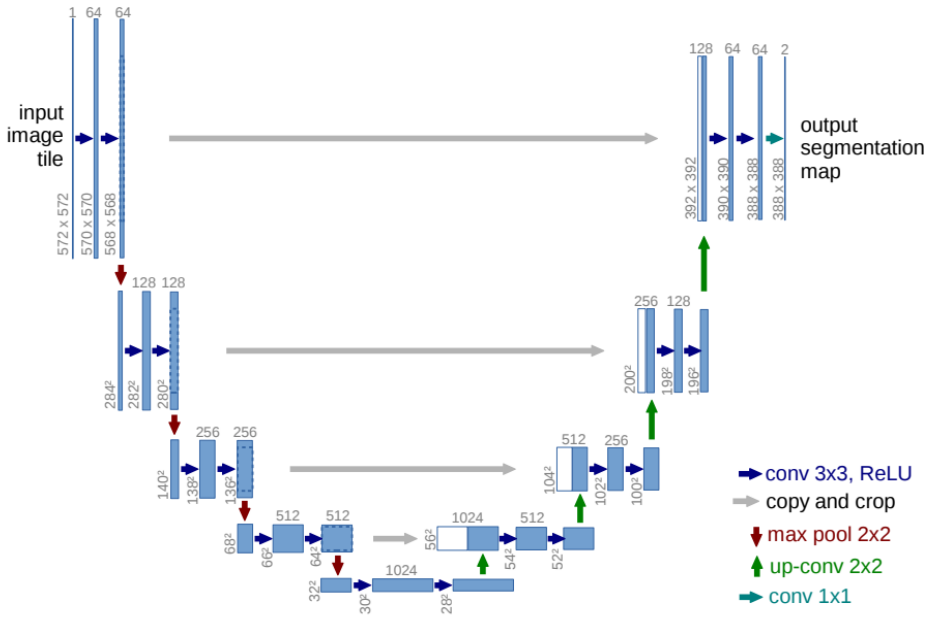

Figure 13: U-Net Architecture

U-Net is comprised of four pooling levels, four upscaling levels, and a very deep level. Each level on the up or down side of the network are made of the same components. Each level of the down side is made of two convolutions with a kernel size of 3x3 followed by a pooling operation with each level doubling the filter count and reducing the height and

width by half. The upwards levels are made of a transposed convolution with kernel size of 2x2 followed by two convolutional layers with a kernel size of 3x3. Each level of the upwards side doubles the height and width, while reducing the number of channels by half. As discussed earlier, the original U-net model does not pad the convolutions along the edges of the image, leading to a reduced output mask size. In every implementation of the model outlined here, the convolutions are padded.

U-Net doubles the amount of convolution filters at each level of the network going down, so by reducing the initial filter count the complexity of the network is reduced significantly. A modification of the U-Net architecture that reduces the parameter by 83% and GFLOPs (Giga Floating-Point Operations) by ¾ is Squeeze U-Net [Beheshti and Johnsson, 2020]. Both methods of reducing complexity were tested and their accuracy metrics compared in **5.7** for total accuracy and in **5.8** for speed.

The complexity and depth of the model is due to the data it was meant to segment, highly complex neuronal structures. The complexity of the images to segment in for this project, in contrast, are simple quadrilaterals. Surrounding the runway though are other features with similar color and shape to the regions of interest. Specifically for Hawk Field, there is a parking lot within 100ft of the runway. Past the runway, there is also a lake that is similar in color to parts of the runway that caused more inaccuracies in the network compared to the parking lot and nearby road.

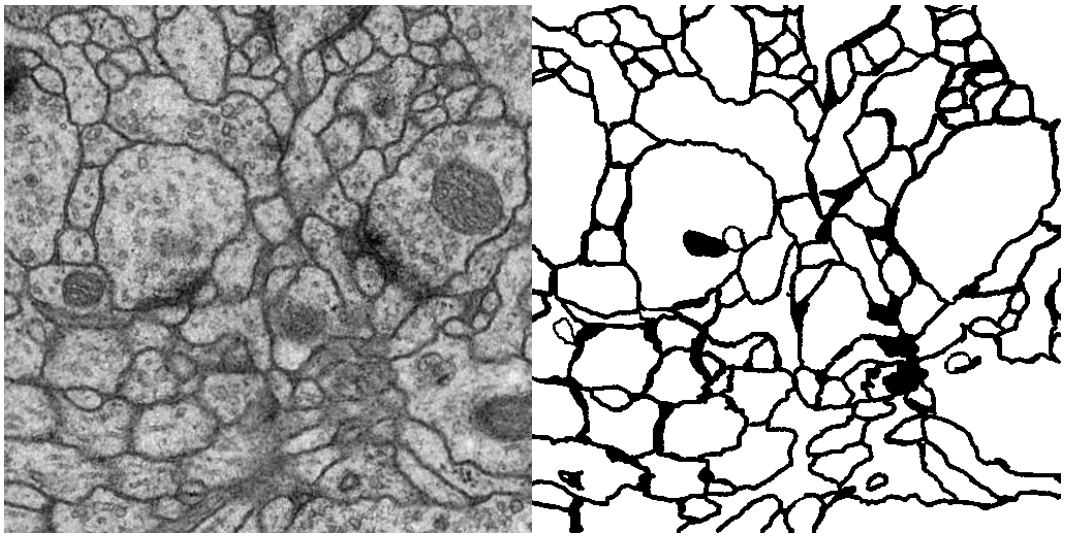

Figure 14: Left is the base image from the ISBI data set, right is the target.

After one of the groundbreaking papers in CNNs, “Very Deep Convolutional Networks for Large Scale Image Recognition” by Karen Simonyan and Andrew Zisserman [Simonyan and Zisserman, 2015] most neural network models traded large kernel sizes for more filters. The increase in accuracy in the CNN model created in [Simonyan and Zisserman, 2015] was attributed to this decrease in kernel size, but increased filters/channels. The basic U-Net model also follows this line of thinking, with the maximum kernel size being the same at 3x3. In “Scaling Up Your Kernels to 31x31: Revisiting Large Kernel Design in CNNs” [Ding et. al., 2022], the decrease of large filter sizes was caused by the [Simonyan and Zisserman, 2015].

In this article [Ding et. al., 2022], there are guidelines for adding large convolutions or replacing small kernels with larger separations. Replacing a single 3x3 kernel by two parallel layers, one with the original 3x3 and one with the new larger kernel size. As shown in that paper, an increase in kernel size from 3x3 convolutions increased

downstream operation accuracy by 3.99%. As an experiment, one modification to U-Net assessed was to change all convolutions with parallel small and large kernel convolutions. The small kernel had a quarter of the original filter count and the large kernel had three quarters**.** The changes in accuracy metrics are shown in **Figure 15** below, with this specific modification of U-Net called "U-Net-L". Due to the main goal of this modification being high accuracy and low computational cost, only low filter counts were tested.

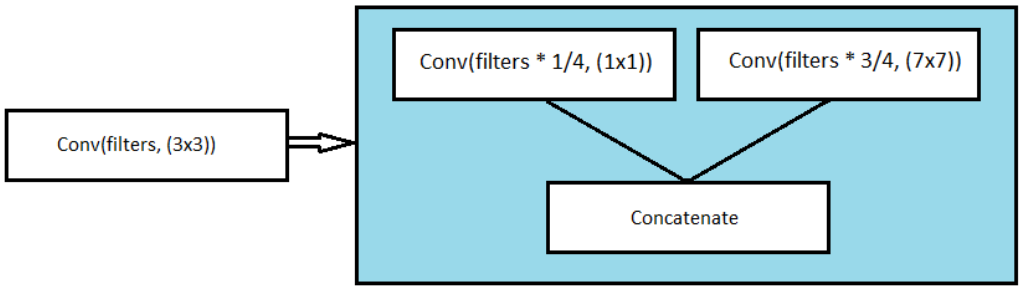

Figure 15: The change in convolution to produce U-Net-Large

U-Net is a powerful semantic segmentation neural network originally used for biomedical imaging. In that use, the speed of prediction is not important so there was an emphasis on accuracy over speed. Speed can be its own sort of accuracy as more and more predictions are made, some of the error can be reduced.

One such technique uses an algorithm called a Kalman Filter [Kalman, 1960]. A Kalman Filter attempts to reduce the error caused by inaccuracies in multiple sources of information. For example, the altitude value given by the autonomous control system mentioned above, ArduPilot, uses barometric pressure and GPS to estimate the altitude above the reference point.

If a lowering of the complexity of the neural network model leads to slightly lower accuracy but large increases in prediction speed, then the final value might be more accurate when combined with other sources. In the **Chapter 12 Future Work**, the implementation of a Kalman filter is discussed.

5.2 Squeeze U-Net:

Squeeze U-Net is the most changed version of the U-Net architectures tested. Each 2D convolution is replaced by 3 convolutions in both the encoding and decoding levels. Squeeze U-Net as proposed in "Squeeze U-Net: A Memory and Energy Efficient Image Segmentation Network" by Beheshti and Johnsson [Beheshti and Johnsson, 2020] splits the convolution into something called a fire module. The fire module is shown in **Figure 16** below.

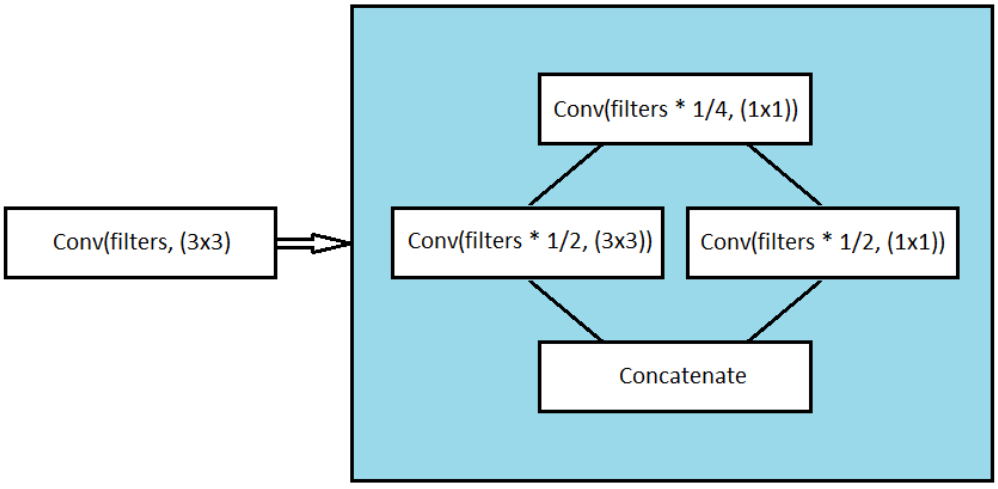

Figure 16: Fire module of Squeeze U-Net

A 1x1 convolution with a quarter of the output filters followed by parallel convolutions with half the output filters, each with a different kernel size, followed by a concatenation along the channel axis for the final output dimensions.

On the up slope of Squeeze-Net, the transposed convolutions are replaced by the transposed fire module which is shown in **Figure 17**:

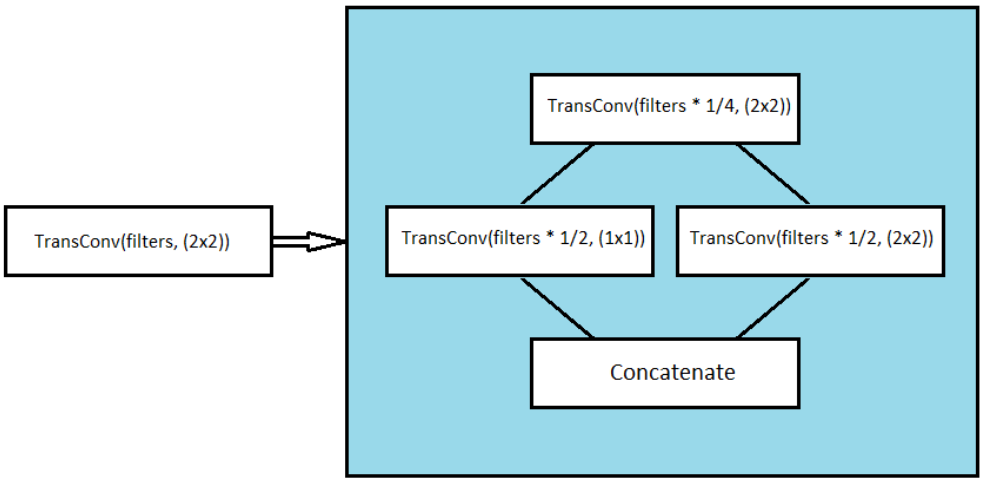

Figure 17: Transposed Fire module of Squeeze U-Net

5.3 U-Net Datasets

All networks were trained on 340 different images over 400 epochs, with a batch size of two. This small of a batch size is somewhat unusual for neural network training, as the network applies a gradient per batch. The gradient is the movement in the weights and biases based on loss that create a more accurate model. A large batch size means the gradient better represents the total dataset.

The training was done on a GTX 1060 using the graphics card application programming interface (API) Compute Unified Device Architecture (CUDA) developed by Nvidia. The machine learning package used, Keras with a TensorFlow backend, has built in CUDA support. While the GTX 1060 is quite a powerful graphics cards compared to cards from a decade ago, it is still low on VRAM at 6GB. It was not capable of loading both the base U-Net network and running more than two images through during training. Some of the reduced size networks would allow for larger batch sizes, but for equivalent loss and training statistics a static batch size was kept.

The small batch size of two images leads to a reliance on a good optimizer, which is the method of applying the gradient of the stochastic objective function (loss) to move the weights/biases. The best performing optimizer was NAdam, a modification on Adam optimization. NAdam was developed by Tim Dozat and outlined in the paper "Incorporating Nesterov Momentum into Adam" [Dozat, 2016]. This selection of optimizer and a short look into NAdam is shown in **5.4 U-Net Optimizer.**

A particularly useful method provided by Keras is called ImageDataGenerator. ImageDataGenerator is integrated into the training method to provide the ability to augment the data through rotation, scaling, shifting, and shearing. This increases the total amount of labeled data significantly without additional human labeling. Another benefit of ImageDataGenerator is that it loads from disk without the need to keep all images in memory. Only the current batch is loaded into VRAM.

There are 112 validation images that were used for all the accuracy comparisons. The validation images are similar in diversity to the training images. The models have never trained on the validation images.

5.4 U-Net Optimizer

Nesterov-accelerated Adaptive Moment Estimation (NAdam) is a powerful optimizer that attempts to increase trained model accuracy, not through architecture, but by modifying how the parameters are changed. In, NAdam the total change of the parameters is based on the next timestep momentum decay factor and the current momentum factor. In previous implementations of momentum, the momentum of the current timestep is based on the previous step momentum and the current step decay factor. NAdam provided a sizable accuracy increase compared to stochastic gradient descent and Adam for in this project but these tests are not displayed.

The learning rate was slowly decreased during the 400 epochs, starting at .0001 until epoch 200 where it was multiplied by $e^{-.02}$ each step leading to a final learning rate of .0000018 at epoch 400. A decrease in learning rate possibly stops the over application of gradients based on the current batch. If the movement on one batch was too large, then the next batches will have lower accuracy due to overfitting on the current batch. The learning rate changes the effect of the gradient on the previous weights. Along with less movement per batch, the low learning rates might also stop the network from skipping from either side of an optimal loss.

5.5 U-Net Losses and Accuracy Metrics

For neural networks, the loss function and accuracy metrics are closely related, especially in sematic segmentation. The loss functions are what determines how accurate the current model is during training and the direction the model needs to change the parameters to be more accurate. Common terms used in these metrics are true positive (TP), true negative (TN), false positive (FP), and false negative (FN). True positives are when the prediction is positive, and the label is positive. True negatives are when the prediction is negative, and the label is negative. False positives are when the model predicts positive, but the label is negative. False negatives are when the model predicts negative, but the label is positive.

With $\hat{y}$ as the mask predicted by the neural network and y as the truth value (human labeled mask) of the target for the model, the mathematical representation of these terms when the prediction and the labels are binary are as follows:

$$TP = (\hat{y} \cap y) \quad \text{[EQ3]}$$

$$TN = (1 - \hat{y}) \cap (1 - y) \quad \text{[EQ4]}$$

$$FN = (1 - \hat{y}) \cap y \quad \text{[EQ5]}$$

$$FP = \hat{y} \cap (1 - y) \quad \text{[EQ6]}$$

True positives are the intersection between the prediction $\hat{y}$ and the label y. True negatives are the intersection between the inverse of the prediction and the label, which is where they both are zero. False positives occur when the prediction value is 1 and the

ground truth is 0, so the union between the prediction and the inverse of the truth. False negatives occur when the prediction is 0 and the truth is 1, so the union between the inverse of the prediction and the truth.

These common terms are a more nebulous when the prediction is only a probability of a pixel belonging to a class. If a pixel prediction is .2, and the label is 0, should that be a positive or negative prediction? In all cases of intersection, the operation conducted is an elementwise multiplication of the two images. These terms transform to

$$TP = \hat{y} * y \quad \text{[EQ7]}$$

$$TN = (1 - \hat{y}) * (1 - y) \quad \text{[EQ8]}$$

$$FN = (1 - \hat{y}) * y \quad \text{[EQ9]}$$

$$FP = \hat{y} * (1 - y) \quad \text{[EQ10]}$$

Different loss functions allow for tweaking what the model is the best at, such as specifying it be strong in determining positives. That is, the false positive rate is low. The loss function can also punish the model when the prediction is negative, but the label is positive. In the predicting the occurrence of a tumor, if the false negative rate is high, many tumors go by undetected. If the false positive rate is high, the only thing wasted is physician time. So, if a supervised learning model is predicting if a tumor exists, a high false positive rate and low false negative rate is quite more acceptable than the inverse.

In “A survey of loss functions for sematic segmentation” by Jadon [Jadon, 2020] several different loss types are looked at for training a similar network on the detection of tumors in brain scans. The performance of the loss functions was determined using the Dice Coefficient, sensitivity, and specificity.

In the terms outlined above, the Dice Coefficient, sensitivity and specificity are:

$$Dice\ Coefficient = \frac{2TP + s}{\left||\hat{y}|\right| + \left||y|\right| + s} \quad \text{[EQ11]}$$

$$Sensitivity = \frac{TP + s}{TP + FN + s} \quad \text{[EQ12]}$$

$$Specificity = \frac{TN + s}{TN + FP + s} \quad \text{[EQ13]}$$

Where s is a smoothing factor applied to prevent division by zero or run off to infinity when the sum of the prediction and label is zero. The smoothing factor is an arbitrary number chosen to be ten in this case.

The Dice Coefficient is a measure of twice the intersection divided by the total number of positives. Interestingly, the Dice Coefficient cannot be used as a loss metric in the way that some of the other accuracy metrics can. In the following page a different metric like the Dice Coefficient, Intersect Over Union IoU, can be directly translated to loss by subtracting the metric from 1. However, this is not possible to for the Dice Coefficient due to a principle called the triangle inequality. The triangle inequality states that the

largest side of a triangle must be less than the sum of the two smaller sides. Intuitively, this is because if the larger side was equal to the sum of the smaller, then there would a single line (try to picture the angle between the two small sides as growing larger and larger finally equaling 180° when the sum is equal) [Gallagher, 2004].

In the case of the Dice Coefficient, this translates to meaning the Dice distance (1-Dice Coefficient) between two points is smaller than the sum of the distance of the two points to a third. Distances measure the distance from the prediction to the label, and therefore must satisfy the triangle inequality.

The loss function that performs the best in the most metrics in the article mentioned above [Jadon, 2020] is called Focal Tversky Loss, which is given as:

$$Focal\ Tversky\ Loss = \left(1 - \frac{TP + s}{TP + \alpha * FP + (1 - \alpha) * FN + s}\right)^{\gamma} \quad \text{[EQ14]}$$

Where α and β are fractions to increase or decrease the training focus on false positives and false negatives, respectively.

Another common form of loss/accuracy metric is the Jaccard Index, also known as intersect over union (IoU). IoU measures the model's ability to predict the correct positive, over the total number of positive pixels between the two models. IoU is calculated using this equation:

$$Intersect\ over\ union = \frac{(\hat{y} \cap y)}{(\hat{y} \cup y)} \quad \text{[EQ15]}$$

$$IoU\ as\ loss = 1 - \frac{TP + s}{\left||\hat{y}|\right| + \left||y|\right| - TP + s} \quad \text{[EQ16]}$$

The most ubiquitous loss function found in almost all areas of machine learning is binary cross-entropy.

$$Binary\ Cross - Entropy = \frac{1}{P}\sum y * \log(\hat{y}) + (1 - y) * \log(1 - \hat{y}) \quad \text{[EQ17]}$$

This is the element wise sum of the label multiplied by the log of the prediction plus the inverse label multiplied by the log of the inversed prediction all divided by the total pixel count. Binary cross-entropy is a strong loss function but does not give any special weight to false negatives or false positives such as in the Focal Tversky Loss or the intersection of the label and prediction such as the IoU loss.

The training method for Keras neural network models allows for the monitoring of not just loss, but also other metrics. All models trained using either binary cross-entropy or IoU loss were equally accurate between while training using the Focal Tversky Loss saw a marked increase in IoU and binary cross-entropy loss (and a corresponding decrease in accuracy). Since the network is trying to minimalize loss, Focal Tversky was the worst for training out of the three loss methods tested. Factors that could have influenced the outcome of Focal Tversky Loss are the two changeable parameters α and γ. α determines the relative weighting of false positives and false negatives while the γ factor changes the

profile of the loss, such as focusing on misclassified low accuracy predictions [Abraham, Khan, 2018**].** This paper by Abraham and Khan lists a γ factor of 4/3 to be the best value, so that was the one used. Different γ might have led to better performances.

5.6 Prediction Thresholding

Another important aspect of the predictions is a threshold over which the output mask is a 1 or a 0 when lower. The final output values are from a sigmoid function, which regularizes the output to between 0 and 1. A sigmoid function at the end of a neural network attempts to change the output to be the probability that a given value is positive. Different probability thresholds lead to better or worse segmentation metric accuracy. If the threshold is too high, many positive predictions are left out. If the prediction is too low, many negatives are counted as positives. For all models, varying probability thresholds were tested on the training images to produce values that maximized the IoU on the training set. No readily available articles outlined a method of smartly calculating a threshold value and most online segmentation models used a simple threshold of .5. This value is based on the center of the sigmoid function. **Table 1** shows the accuracy for the U-Net models with set threshold of .5 and with the best threshold determined from the train dataset. The metric values are from the validation set. **Table 2** show the same information but for the Squeeze U-Net models.

Table 1: Accuracy metrics of U-Net models with .5 threshold and best threshold based on training data. Metrics are from validation data. Better of same model in bold. Number is the count of initial filters.

Model	*t*	Min IoU	Avg IoU	Min Sensitivity	Avg Sensitivity	Min Specificity	Avg Specificity
U-Net 64	.5	0.6181	.9496	**0.8330**	**0.9834**	0.9033	0.9902
U-Net 64	.7	**0.7307**	**0.9518**	0.7576	0.9680	**0.9296**	**0.9951**
U-Net 16	.5	**0.7163**	**0.9572**	**0.7271**	**0.9819**	0.9060	0.9924
U-Net 16	.6	0.6057	.9550	0.6090	0.9694	**0.9268**	**0.9947**
U-Net 8	.5	0.3967	0.8879	**0.6358**	**0.9709**	0.9498	0.9911
U-Net 8	.7	**0.4622**	**0.9059**	0.5037	0.9439	**0.9709**	**0.9958**
U-Net-L 8	.5	**0.0097**	**0.7033**	0.0002	**0.7249**	0.9698	0.9964
U-Net-L 8	.6	0.0043	0.6779	0.0002	0.68203	**0.9716**	**0.9975**

Table 2: Accuracy metrics of Squeeze U-Net models with .5 threshold and best threshold based on training data. Metrics are from validation data. Better of same model in bold.

Model	*t*	Min IoU	Avg IoU	Min Sensitivity	Avg Sensitivity	Min Specificity	Avg Specificity
Squeeze 64	.5	0.5263	**0.9599**	**0.7470**	**0.9796**	0.9297	0.9958
Squeeze 64	.7	**0.5587**	0.9512	0.6877	0.9617	**0.9490**	**0.9978**
Squeeze 16	.5	**0.6666**	0.9447	**0.7215**	**0.9739**	0.9472	0.9739
Squeeze 16	.6	0.5164	**0.9484**	0.5173	0.9560	**0.9534**	**0.9970**
Squeeze 8	.5	**0.1926**	0.8822	**0.4921**	**0.9673**	0.9366	0.9877
Squeeze 8	.7	0.1871	**0.9102**	0.3842	0.9510	**0.9532**	**0.9932**

In general, the change in the metrics is expected. With a higher threshold set, the pixel probability must be higher to be considered a positive leading to more false negatives and less false positives. Sensitivity is the ratio of true positives over total labeled positives and will fall if more false negatives are present. The opposite is true for specificity, the ratio of true negatives to total negatives. However, the intersect over union metric has false positives and false negatives at the same weighting. For final accuracy rating, the threshold with the highest average IoU will be used and listed in the tables and graphs.

5.7 Accuracy of U-Net Models

Table 3 below shows the accuracy for each model with the calculated threshold. Keep in mind, the threshold was not based on the validation data, rather the training data, and compared to the baseline .5 threshold on the validation set. There would be some cherry picking if the best accuracy was chosen. However, since the best accuracy on the training set in all but two cases was the non-standard threshold, that non-standard value was the threshold used.

Table 3: Final U-Net accuracy metrics. Number is the count of initial filters.

Model	*t*	Min IoU	Avg IoU	Min Sensitivity	Avg Sensitivity	Min Specificity	Avg Specificity
U-Net 64	.7	**0.7307**	0.9518	**0.7576**	0.9680	0.9296	0.9951
Squeeze 64	.7	0.5587	0.9512	0.6877	0.9617	0.9490	**0.9978**
U-Net 16	.6	0.6057	**.9550**	0.6090	**0.9694**	0.9268	0.9947
Squeeze	.6	0.5164	0.9484	0.5173	0.9560	0.9534	0.9970
U-Net 8	.7	0.4622	0.9059	0.5037	0.9439	0.9709	0.9958
Squeeze 8	.7	0.1871	0.9102	0.3842	0.9510	0.9532	0.9932
U-Net-L 8	.6	0.0043	0.6779	0.0002	0.68203	**0.9716**	0.9975

Interestingly, the accuracy of the simpler models is a bit higher on the validation data. U-Net 16 Filters is higher in most metric averages when compared to the larger networks. It is important to note that the total parameter count is significantly less between 16 and 64 initial filters. This could be due to over-fitting on the train data. Over-fitting is the tendency of deep neural networks to learn the specific training images and reproduce the result based on the actual image, as opposed to the object in the image. Instead of recognizing objects, an overfitted network recognizes images. These models do include dropout layers with the images being augmented through rotation, scaling, and translation, which helps to reduce overfitting.

5.8 Model Complexity Comparisons

The models will be measured by four metrics: parameter count, floating-point operations, one image inference speed, and two image inference speed. Parameters are the values used by the network to connect one layer to the next. In the case of U-Net, all parameters

are trainable in the form of weights and biases. FLOPs stand for floating-point operations, which is the needed number of operations per inference. Not to be confused with FLOPS (note the capital "s") that counts the number of floating-point operations per second, usually used to measure the speed of a GPU or CPU. The true speed of a single image through the network will be slower than expected given the GPU FLOPS and the network FLOPs as there is time added for the transfer of data to the hardware. The model was loaded into memory and ran a prediction prior to the timed inferences to prevent any GPU warm up time from being counted. These times do include the memory transfer for the image from RAM into VRAM.

Table 4: Performance Metrics

Model	FLOPs (Billions)	Parameters	1 Image Time	2 Image Time
U-Net 64 Initial Filters	385.53	31,031,745	.1549	.2741
Squeeze 64 Initial Filters	138.59	5,740,577	.1183	.1966
U-Net 16 Initial Filters	33.19	2,006,529	.0768	.1124
Squeeze 16 Initial Filters	34.29	360,713	.0853	.1042
U-Net 8 Initial Filters	11.34	524,065	.0671	.0957
U-Net-L 8 Initial Filters	3.23	199,365	.0867	.1052
Squeeze 8 Initial Filters	2.29	135,341	.0870	.1075

The performance gain from less FLOPs and less parameters has a diminishing return, as seen from the table above. The U-Net model with 8 initial filters has near four times the number of parameters and FLOPs as the same initial filter count Squeeze U-Net, but with less processing time. There are many reasons for this plateau of speed. The primary

speculation is due to the transference of the memory from VRAM into caches. If the total operations are decreased by 80% but there is not a considerable decrease in operation time, what could the cause be? FLOPs should be the surest measure of operation speed. This moves from personally programmed software efficiency (parameter count, operations) to hardware efficiency (transfer from VRAM into L1/L2 cache) and optimizations performed by the graphics card drivers/ API (cuDNN/CUDA).

The actual cause of this peak operation speed is not explored in this paper, and there was little research available for this problem. A possible indication of this plateau is shown in the faster inference speed of U-Net compared to the Squeeze U-Net. Squeeze U-Net should operate faster, as shown in the FLOP count, but the inference time is slower. This might be due to all convolutions changing to two parallel convolutions with less total parameter count. Another indication is the difference between one image and two images inference time. In the smaller NNs the inference time is not doubled, possible due to the low memory allowing for more parallel computations.

5.9 U-Net conclusions

Surprisingly, the most accurate version (IoU accuracy) of all models evaluated was the U-Net with 16 initial filters. The operation speed of this model is also quite fast at .0768s per image. With a frames per second of 13.02 and IoU average of .9550, the model is more than capable for the task outlined in this paper. Moving forward, all downstream tasks and operations will be done using this model.

The worst image for all models fell into one of two types, shown below.

Figure 18: Worst validation image type for IoU accuracy (Left) and U-Net 16 prediction (Right)

Figure 19: Worst validation image type for IoU accuracy (Left) and U-Net 16 prediction (Right)

5.10 Resources used for U-Net Models

The U-Net model was completely programmed in Python using the Keras machine learning package with TensorFlow backend. The flexibility and changeability of Keras with TensorFlow is almost unrivaled. The entirety of U-Net could be written in less than 30 lines. In the code found in the Appendix, the model was expanded to ~70 lines for

readability. Image loading and resizing was performed using the Python Image Library (PIL). All other matrix manipulation used NumPy.

CUDA and cuDNN created by Nvidia allow for the neural networks to train and predict using graphics cards. The change in speed from CPU to graphics cards is incredible, from one second per image to less than a tenth in most cases. As mentioned above, the graphics card used was the Nvidia 1060 with 6GB of VRAM and core clock of 2037 MHz (35% overclocked).

6 YOLOv3 Model for Runway Location

6.1 YOLOv3 Overview

YOLOv3 [Redmon and Farhadi, 2018] is a real time object recognition system based on a residual convolutional neural network. The purpose is extremely fast prediction times on a wide range of classes, with some sacrifices for accuracy. As seen in **Figure 20**, the speed of YOLOv3 is higher when compared to other contemporary methods.

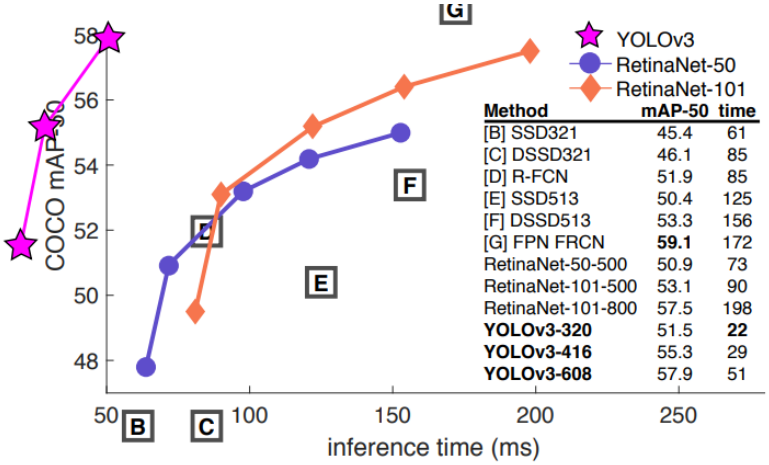

Figure 20: Accuracy for YOLOv3 compared to contemporary models on COCO dataset. Graph from [Redmon and Farhadi, 2018]

YOLOv3 is now out of date, with no public update in the past few years, but it remains one of the better architectures with a significant amount of resources online. The original creator of YOLO, Joseph Redmon**,** stopped development due to growing concerns of the military applications. There are continuations of YOLO including a 4th and 5th generation, but the information online is limited and Redmon is not a contributor.

YOLOv3 is based on a residual convolutional neural network called DarkNet-53. Residual neural networks were proposed to reduce the degradation of accuracy associated with higher model depth [He et. al., 2015]. As depth increases, accuracy will increase as well up to a certain model complexity. As more layers are added accuracy will

continually increase until a certain level of complexity, where the accuracy will sharply decrease. On overly complex models, lower accuracy on the validation set is quite common due to overfitting. However, decrease in accuracy that residual layers combat is also seen in the training set.

Residual layers combat this by feeding an additional layer output into convolutional blocks a few steps away from the next block. This partially bypasses the blocks in the network creating an artificially shallow neural network while also keeping a very deep path. The image in **Figure 21** below shows the full shape of the residual network YOLOv3 is based on

	Type	Filters	Size	Output
	Convolutional	32	3 × 3	256 × 256
	Convolutional	64	3 × 3 / 2	128 × 128
1×	Convolutional	32	1 × 1	
	Convolutional	64	3 × 3	
	Residual			128 × 128
	Convolutional	128	3 × 3 / 2	64 × 64
2×	Convolutional	64	1 × 1	
	Convolutional	128	3 × 3	
	Residual			64 × 64
	Convolutional	256	3 × 3 / 2	32 × 32
8×	Convolutional	128	1 × 1	
	Convolutional	256	3 × 3	
	Residual			32 × 32
	Convolutional	512	3 × 3 / 2	16 × 16
8×	Convolutional	256	1 × 1	
	Convolutional	512	3 × 3	
	Residual			16 × 16
	Convolutional	1024	3 × 3 / 2	8 × 8
4×	Convolutional	512	1 × 1	
	Convolutional	1024	3 × 3	
	Residual			8 × 8
	Avgpool		Global	
	Connected		1000	
	Softmax			

Figure 21: YOLOv3 Backbone ResNet [Redmon and Farhadi, 2018]

6.2 YOLOv3 Implementation

To run YOLO at peak performance on the same computer as the U-Net models was difficult. The original version of Darknet-53 (YOLOv3 ResNet backbone) was developed on and for native Linux systems, which the testing computer was not. Performance issues occurred when using a virtual machine to test, so a Windows version of Darknet must be used. This was provided through the GitHub of Alexey Bochkovskiy. He is not the only person working on the project, but he is the owner of the repository and the primary contributor.

The YOLO version was trained over 6800 epochs on the same training images as U-Net. The training speed was staggering. The total training time took 4 hours which is near the same time as 400 epochs on the faster U-Nets, with almost 50 times the total parameter count.

YOLOv3 on the validation images has a IoU accuracy of .8846 with a time of inference of .048 seconds. This is twice as fast as the U-Net speed plateau shown in **5.8.** However, there is a decent decrease in IoU accuracy from the U-Net 16 accuracy of .9550. The YOLOv3 prediction is only the box around the runway, so additional image processing techniques would also need to be employed to gather the extra information already provided by the segmentation of U-Net. **Figure 22** below shows what the YOLOv3 predictions look like.

Figure 22: YOLOv3 Prediction boxes

6.3 YOLOv3 Conclusions

While the YOLO model does have a significant speed increase compared to U-Net, the decrease in accuracy and the needed post inference operations are quite significant. The plan was to use Canny edge detection, followed by a Hough Transform, to get the runway edges. From experience, this method can lead to large error due to the hundreds of different lines possible through edge points when using the Hough algorithm. For these reasons, all additional content in this paper is based on the predictions performed by U-Net with 16 initial filters.

7. Recovery of Runway Shape

7.1 Background

With the segmented image, the next task is to extract the shape from the binary prediction. Expectedly, determining the four corners of a quadrilateral from a point cloud is non-trivial. During target labeling, the corners could be placed outside of the image dimensions, allowing four corners to describe a more complex shape, but now the corner count for the runway is not explicitly four. A partially visible runway can be inscribed by three corners up to eight corners. The runway in the image will have eight corners when all four true corners are cut out of the image, when the camera is pointed straight down, and the runway runs diagonally along the image.

The irregular shape of the prediction is called a contour. To move from the point cloud prediction to a contour a method of edge following must be employed. The contour of the prediction will be highly irregular, since in real life the runway is certainly irregular. A method to move from the irregular contour to a simpler shape will be employed. Additionally, bad predictions might lead to additional segmented regions in the prediction mask. The region describing the runway must be separated from other regions in the prediction.

7.2 Point Cloud Prediction to Runway Contour

The edge pixels of the segmented regions are found using the chain method for edge following. Chain edge following uses a binary array as the input, which is the format of the prediction after a probability threshold. The chain algorithm parses the prediction

mask from left to right moving down after each successive row of zeros, until a one is found.

After the first one is found, the primary chain method can start, which is a system of left and right turns based on the value encountered. If a one is found, turn left. If a zero is found, turn right. Following a turn, a movement occurs in the new direction. A location is determined to be an edge depending on the order of turns and movements to get to that location. If the turn on the current location is right, and the previous was left, then the location the left turn was performed on is a boundary. Another boundary case is after a right turn from a zero to a one value, the one location is a boundary. The ordering of turns disregards any interior one value.

Contouring also allows for disregarding regions that are not of interest. While the predictions of the neural network are generally good for the runway, in some cases there are more than one region predicted**.** In all encountered cases, if the runway is in the image, then it was the largest contour area. Interestingly, the contour points found using the chain method are in clockwise order. Due to this, Shoelace theorem can be applied to calculate the areas of the contours without any additional steps.

7.3 Contour to Shape

While the edge pixel values are quite valuable and can be fed into the location approximation methods, the shape is still unknown. To recover altitude from a known runway through known dimensions, that dimensions must be reconstructed. In the case

outlined in **Chapter 8** the known dimension is the width of the runway. Since the contour approximation gives all edge values of the runway, there is no indication of what edge pixels are at the same location on either side of the runway to approximate the width in the image. To approximate the pixel width, the shape of the runway must be recovered.

With the contour of the region highly likely to be the runway, the next step is to get the least complex shape that circumscribes the region. In many cases this will be a quadrilateral since the runway itself is a quadrilateral. As mentioned above, the runway can take on a complex shape involving as little as three corners, up to a maximum of eight. At no point was an image where the runway is described in eight corners encountered due to low pitch values while recording, but it is still considered in the program.

Ramer–Douglas–Peucker [Douglas and Peucker, 1973] algorithm attempts to reduce the number of vertices in a polyline (line described by multiple points) by first determining the point furthest from the line formed by the two polyline endpoints. If that distance is larger than a parameter ϵ, the point is kept, and the algorithm then runs again twice with the start/furthest as end points and furthest/end as end points. If the furthest point from the endpoints is less than the input parameter away from the end points line, then that point can be disregarded. The Ramer–Douglas–Peucker algorithm operates recursively, separating the polyline into smaller and smaller segments until the furthest distance is less than ϵ.

The implementation of the Ramer–Douglas–Peucker algorithm used was provided by OpenCV. This version can also use closed shapes (such as a chain contour) as the polyline through setting the two points most separated points on the contour as the initial start and end points for the algorithm.

To return reliable results, the parameter ϵ that determines the maximum distance needs to be changed. If the parameter is too large, the final polyline shape might be a terrible representation of the original polyline. If ϵ is too small, then too few vertices will be removed. The size of the runway in the image also changes, so a set ϵ might be good in one instance but poor in another. To combat this issue, ϵ is not a set value but instead is based on the perimeter of the contour. Parameter ϵ values are also looped through for a few values before an image is disregarded. If the Ramer–Douglas–Peucker algorithm cannot produce the number of corners that describes runway shapes, the image is disregarded. The number of vertices in an image can range from 3 (when only one true corner is visible) to 8 (when most of the runway is visible except for the true corners).

Since the input into the Ramer–Douglas–Peucker algorithm is the segmented image of the runway, the edges should be quite true to the quadrilateral. In practice, the segmented images are not perfect and capture part of the taxiway or have complex edges so as the contour lessens in complexity, the first points removed should be these extraneous points.

8 Transformations from Pixels to Real World Locations

8.1 Background

Transferring dimensions in the camera plane to the real world has many issues involved due to the distortions in the image caused by the lens. The distortions must be corrected for, if significant. However, the quality of the camera and sensor in the drone used led to little distortion. **Figure 24** on the next page shows how little distortion was in the image.

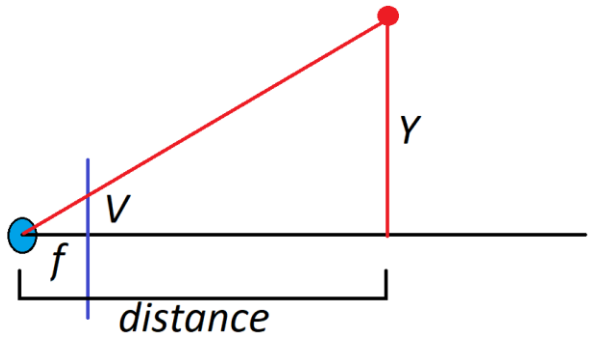

Figure 23: Basic lens model with reversed image plane.

The basic lens equations are quite simple due to the ratios of triangles. As seen in **Figure 23** above, the triangle made from the lens, the point, and the horizontal distance share all angles with the triangle made by the lens, the image plane location, and the focal length. The lens rests on the Z axis with the horizontal distance being the Z distance.

The object vertical location and horizontal location can be calculated using the ratios:

$$\frac{U}{f} = \frac{X}{distance} \quad \text{[EQ18]}$$

$$\frac{V}{f} = \frac{Y}{distance} \quad \text{[EQ19]}$$

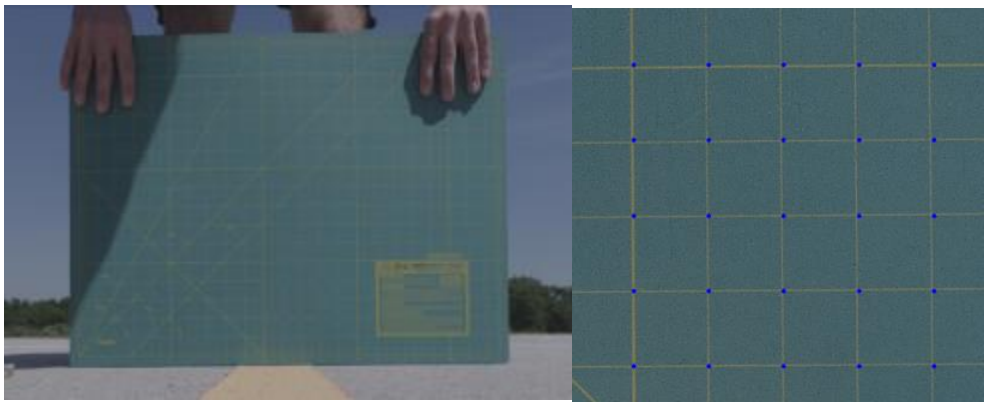

Figure 24: From real world measurements, placing dots on the grid in the image.

Figure 24 above shows this simple method applied to that grid shown above. The small variations might be due to the wind bending the mat or the mat being held at a slight angle.

However, this does not work when there is a difference in angles between the image plane and the plane the object lies on. If the real world is simplified to a flat plane, then the image plane and the object plane will never be parallel unless the camera is pointed straight down.

The first major simplification for these equations is made here: the real world is a flat plane. Discussed later in **Chapter 8.7**, the real world is everything but a flat plane. Even runways are not close to flat. Without this simplification though, the exact terrain data would be needed to calculate pixel-to-real locations.

Since the camera is not pointed straight down, the basic lens ratios are disregarded. To calculate the location now, a pixel is taken to be a line pointing out of the lens. Where this line intersects the real world is the location of the object.

All these points and planes, the pixel location in the image frame, the lens location, and the real plane, must be transferred into a single coordinate system. This system is called the homogenous coordinate system.

8.2 Forming the Homogenous Coordinate System

The image plane is the physical sensor plane at *focal length* from the lens. The homogenous coordinate system is centered at the intersection of the lens centerline path and the real plane. The homogenous coordinates X/Y are parallel to the image plane U/V, while the Z axis is offset. The real plane is rotated from the homogeneous coordinate X/Y plane by 90° - pitch. This is visualized in **Figure 25** (pg. 55), where the line labeled homogenous plane represents the X/Y plane in the coordinate system. **Figure 25** is collapsed along the X axis, meaning the up is Z+, down is Z-, left is -Y, and right is Y+.

Here is the second large assumption: the only rotation of the real plane from the image plane is along the X axis (pitch). A great aspect of the Phantom 4 Pro is the gimbal the camera sits on. The camera was set at an angle from horizontal, and it held that position. Therefore, there should be no roll. While this is somewhat true, the gimbal is not perfectly accurate, and the values are based on the drone's pitch/roll. These two sources

of error mean that a small roll component may exist in the images, but it is disregarded in all calculations. Discussed in chapter **8.7,** the gimbal is lagging a bit as well.

The real plane and lens center intersection is based on the angle of pitch as well as the altitude. Since the altitude is based on the gravitational down, it is considered to make a right angle with the real plane. Therefore, the height of the lens above the intersection between lens center and the real plane is:

$$Height\ of\ Lens = h\ = \frac{Altitude}{\cos\left(90° - pitch\right)} \qquad \text{[EQ20]}$$

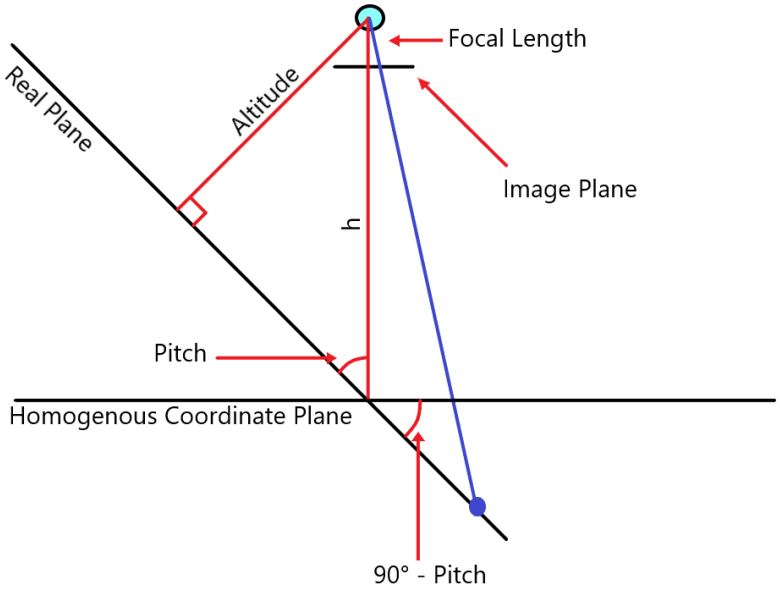

Figure 25: Real plane offset/Homogenous coordinate system

The image above shows the representation of this system in a somewhat easy to read format with a ray from a real point to the pixel point. Due to the complexity of representing 3D objects in a 2D format, some information is lost in this representation. What is shown is the Y axis (left/right) and the Z axis (up/down). All systems, homogenous, real, and image, are collapsed along the same axis U, X_h, and X_r,

respectively. Some items are not labeled due to the relative size of the dimension, such as the pixel value V. (U, V) is where the ray in blue intersects the image plane. The image plane is offset down from the lens by the focal length. The lens height above the homogenous plane is based on the altitude of the drone and the pitch angle. The line labeled *h* in the figure is the lens height, which is along the Z axis. The lens location in the homogenous system is (0, 0, h).

The offset of the image plane from the focus is reversed due to the automatic righting of the image. In a lens, the light is refracted at an angle from the center inverse that of the incoming angle (right is left, up is down). When transferring the sensor information to the image or video, the sensor information is inverted to correct for the lens inversion. Because of this automatic inversion, the image plane can be flipped to be between the lens and the rays.

To calculate the real-world distance to a pixel, first the equation for the real plane in homogeneous coordinates muse be found. This equation is simply based on the rotation from the X/Y plane, that rotation is the pitch. The equation of the real plane is:

$$0 = \cos(pitch) * Y + \sin(pitch) * Z \qquad \text{[EQ21]}$$

This equation is based on a pitch between 0° and 90°, where the angle is the rotation down from the horizontal. In the Phantom 4 Pro, this is a bit different as the pitch of the

gimbal and pitch of the plane are negative for values below horizontal, so those must be multiplied by -1 before using the plane equation and all following equations.

As mentioned above, this equation disregards any roll component. If a roll component must be added, the differences would be seen in the plane equation and the equation for the lens height from the origin.

8.3 Pixel to Real World and back

From the basics of cameras, a ray must enter the lens for it to appear in the image. The ray will also hit a section of the image plane as shown in **Figure 25**. To find the area in the real world that a pixel represents is as simple as finding where ray intersects the real plane.

To find the ray created by a pixel (U, V), first the pixel dimensions must be transferred into distance values. This transfer will depend on the image size and the sensor size. The sensor size of the Phantom 4 Pro is a 1” CMOS. These sensors use an industry standard of 3:2, meaning the size of the sensor in millimeters is 13.2x8.8. At 3:2 the total pixel dimensions are 5472x3648.

The image array starts at the top left corner at (0,0), so the pixel values must not only be scaled to the sensor dimensions. The values must also be offset making (0,0) the center of the image.

To transfer from pixel dimensions to the image plane dimensions:

$$U_i = \frac{U_{pix} * Sensor\ Width}{Image\ Width} - \frac{Sensor\ Width}{2} \quad \text{[EQ22]}$$

$$V_i = \frac{Sensor\ Height}{2} - \frac{V_{pix} * Sensor\ Height}{Image\ Height} \quad \text{[EQ23]}$$

Since the ray created by a pixel at [U, V] must pass through the lens, two points that lie on the ray are [0, 0, h], the lens location, and [U_i, V_i, h - f], the pixel location. Both are in terms of the homogenous coordinate system. A parameterized has the equation:

$$\text{Ray} = \{(X_1 - X_0) * t + X_0, (Y_1 - Y_0) * t + Y_0, (Z_1 - Z_0) * t + Z_0\} \quad \text{[EQ24]}$$

A parameterized ray going from the lens to the image plane will have an equation of:

$$= \{U_i * t, V_i * t, ((\text{h} - f) - h)\ t + h\}$$

$$\{U_i * t, V_i * t, -f * t + h\} \quad \text{[EQ25]}$$

This will intersect the real plane at a given t value. To find the t value, the ray equation is put into the real plane equation.

$$0 = \cos(pitch) * V_i * t + \sin(pitch) * (-f * t + h)$$

$$\sin(pitch) * f - \cos(pitch) * V_i = \frac{\sin(pitch) * h}{t}$$

$$\frac{1}{\sin(pitch) * f - \cos(pitch) * V_i} = \frac{t}{\sin(pitch) * h}$$

$$\frac{\sin(pitch) * h}{\sin(pitch) * f - \cos(pitch) * V_i} = t \quad \text{[EQ26]}$$

This t value is then plugged into the ray equation to find the location of the object in the homogenous coordinate system. A simple rotation and offset are then performed if the location of the object in the real plane is required, but distance from the lens can be calculated using the homogenous coordinates.

This equation is undefined when $\sin(pitch) * f = \cos(pitch) * V_i$ due to division by zero. The undefined portion of the equation relates to where the line never intersects the real plane due to the angle of the ray as shown in the figure below.

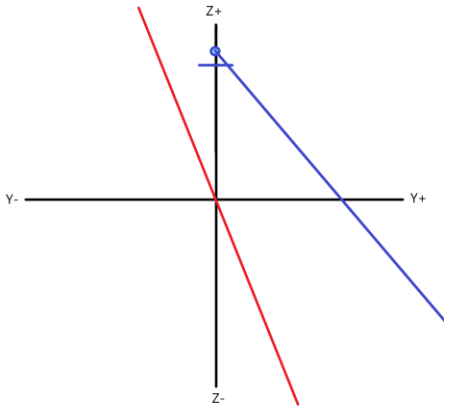

Figure 26: Intersect equation undefined because ray (blue) starting at lens with this direction never intersects the real plane (red)

This method can also be applied in reverse to get the pixel location of a real-world object with the following rotation matrices where θ = 90° - pitch. These matrices are simplified as the real plane and anything on it has been assumed to be flat and every Z value in the real coordinate system is 0.

Rotation matrix from real to homogenous, where Y is already offset by $h * \sin(\theta)$

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos(\theta) & 0 \\ 0 & -\sin(\theta) & 0 \end{pmatrix} [X, Y, 0]^T \quad \text{[EQ27]}$$

And going from homogenous to real:

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & \frac{1}{\cos(\theta)} & 0 \\ 0 & 0 & 0 \end{pmatrix} [X, Y, Z]^T + \begin{pmatrix} 0 \\ h * \sin(\theta) \\ 0 \end{pmatrix} \quad \text{[EQ28]}$$

The reverse method is quite similar as it still involves figuring out the ray equation and where it intercepts a plain.

8.4 Verifying Previous Equations

To assess the method of rays, as much error as possible needs to be eliminated. One source of error eliminated was the altitude. Since the drone relies on GPS and barometric pressure for altitude, the drone value for altitude was disregarded for testing in favor of set locations with known heights. For the most accurate test, the drone was placed on a table with the camera 24.5” above the runway, with tennis balls set at a distance from the drone on the runway. The equations were then tested in both directions, using the known real location of the tennis ball

Using this transfer from real to homogenous coordinates, then running the system listed above the equations were verified to work, at least in a simulation of the camera position, pitch, and real position.

In practice these equations have a certain amount of error, down to human measuring and error in the gimbal and inertial measurement system (IMU). The pitch angle of the gimbal is based on the IMU reading of the drone. If the drone is reading that it is pitched 3° from horizontal and the gimbal is set to keep 20° from horizontal, the actual gimbal

reading will still be 20° no matter the previous movements or the current pitch from horizontal. Discussed more in **Chapter 8.7,** the pitch can be inaccurate.

The gimbal pitch reading will have a combined error based on the gimbal measurements and the IMU measurements, and the lag between drone movements and the gimbal speed. These errors were extensively researched with no clear values for any.

In the **Figure 27** below, the real-world location to pixel tracks the white line completely within the shown pitch values. However, the corners are quite bad possibly due to some of the errors mentioned in **Chapter 8.7.** The distance to the white line was not measured but extrapolated from the space between the yellow lines and the yellow line length. The distance could to the white lines could be longer or shorter if there is significant error in the painting of the center strip.

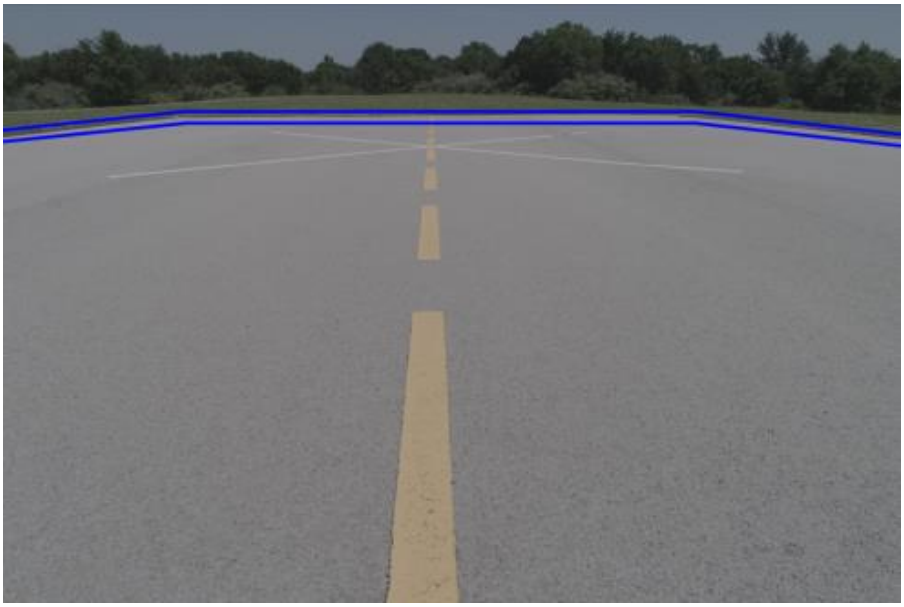

Figure 27: Algorithm tracing the runway outside paint pitch = 19° and 20°

8.5 Altitude calculation from a known runway

For the estimation of the altitude, it is assumed that only the width of the runway is known. It is the dimension most likely to be completely in frame, so it can be used more often if the equations are correct.

The easiest method for this is when there are two points known (or thought) to be the actual corners of the runway. The distance in the real plane between these two corner points should be the known dimensions. So simply:

$$\sqrt{(X_1 - X_0)^2 + (Y_1 - Y_0)^2} = width$$

The issue is that all equations are already in terms of *h*, or the lens distance from the real plane. Plugging in the values from above for X/Y in the real plane. The $h * \sin(90 - pitch)$ offset term cancels out for the Y values.

$$\sqrt{(U_{1i} * t_1 - U_{0i} * t_0)^2 + \left(\frac{V_{1i}}{\cos(90 - pitch)} * t_1 - \frac{V_{0i}}{\cos(90 - pitch)} * t_0\right)^2} = width$$

For ease of reading, the t values are compressed to the following values. Check [EQ26] for t derivation:

$$M_1 * h = t_1$$

$$M_0 * h = t_0$$

And with these values plugged in:

$$\sqrt{(h * (U_{1i} * M_1 - U_{0i} * M_0))^2 + (h * \left(\frac{V_{1i}}{\cos(90 - pitch)} * M_1 - \frac{V_{0i}}{\cos(90 - pitch)} * M_0\right)^2}$$

$$= width$$

Moving *h* outside of square term:

$$\sqrt{h^2 * (U_{1i} * M_1 - U_{0i} * M_0)^2 + h^2 * \left(\frac{V_{1i}}{\cos(90 - pitch)} * M_1 - \frac{V_{0i}}{\cos(90 - pitch)} * M_0\right)^2}$$

$$= width$$

Moving *h* outside of square root

$$h * \sqrt{(U_{1i} * M_1 - U_{0i} * M_0)^2 + \left(\frac{V_{1i}}{\cos(90 - pitch)} * M_1 - \frac{V_{0i}}{\cos(90 - pitch)} * M_0\right)^2} = width$$

Solving for *h:*

$$h = \frac{width}{\sqrt{(U_{1i} * M_1 - U_{0i} * M_0)^2 + \left(\frac{V_{1i}}{\cos(90 - pitch)} * M_1 - \frac{V_{0i}}{\cos(90 - pitch)} * M_0\right)^2}} \quad \text{[EQ29]}$$

The other case is when none of the true corners are the ones determined from the segmented image. This occurs when the drone is too close to the runway with the camera angled too far. Ideally, the runway edge lines will be parallel, so the distance between the two lines is the width of the runway.

If two lines of the runway are known, it is difficult to determine what two points to compare that are the width apart. However, the runway lines *should* still be parallel in the real plane. So, the width is the distance between the two lines.

In the following equations, the M11/M12 refer to the parameters for the pixel rays in the same form as above (lens distance removed) for the first line and M21/M22 for the second line.

$$slope\ 1 = \frac{(Y_{12} - Y_{11})}{(Y_{12} - X_{11})}$$

$$slope\ 2 = \frac{(Y_{22} - Y_{21})}{(Y_{22} - X_{21})}$$

The parameter b from the equation in the form y = slope * x + b must be calculated:

$$b_1 = Y_{12} - slope\ 1 * X_{12}$$

$$b_2 = Y_{22} - slope\ 2 * X_{22}$$

With the equations for the real values from image plane values:

$$b_1 = \frac{V_{12} * M12 * h}{\cos(90 - pitch)} + \sin(90 - pitch) * h - slope\ 1 * U_{12} * M12 * h$$

$$b_2 = \frac{V_{22} * M22 * h}{\cos(90 - pitch)} + \sin(90 - pitch) * h - slope\ 2 * U_{22} * M22 * h$$

Each term in these equations has an *h* component, so it can be factored out in the next equation. Then the distance between the two lines is calculated using the equation below:

$$known\ width\ = \frac{h * \mid b_2 - \ b_1 \mid}{\sqrt{1 + m^2}}$$

$$lens\ height\ = \frac{known\ width * \sqrt{1 + m^2}}{\mid b_2 - \ b_1 \mid} \qquad \text{[EQ30]}$$

With pixels generated from the move from real → pixel, the altitude is recovered perfectly. When the pixels are selected by hand to be along the runway line in the image, the altitude determination has a decent amount of error. The slopes of the lines are not equivalent, meaning the lines are not parallel. There are then three altitudes calculated, one for each slope and one for the average of the two. The individual slope calculations have errors greater than 1000% but using the average the error is reduced to less than 25%. In **Chapter 8.7**, the error is explored a bit more in depth.

8.6 Techniques that have been used by other researchers for the research problem

During research, other methods of turning pixel locations into real world locations were explored. These methods relied on inversing the camera matrix. The camera matrix transforms real world dimensions into pixel dimensions based on rotation, scaling, and offset. The camera matrix and its inverse could be used for these operations, but the equations for altitude from known locations become quite complex. By leaving the transformation from real to pixel, and pixel to real in the form described above, the equations are simplified. Of course, these equations only work for a singly rotated plane along the camera X axis. The equations become difficult for a doubly rotated camera

The idea for a ray-based system was found in a few various places online, but never flushed out into full equations based on planes such as it is laid out here. A homogenous coordinate system based on the intersection of the camera center line and the real plane is also novel.

8.7 Error with Methods

These methods certainly have error, mainly due to the in-exactness of the runway and the pitch. The runway is not a flat plane but has hills and valleys within the length. **Figure 28** below shows how one end of the runway is curved as well as cupped. The cupping of the runway will lead to errors where the ray generated by a pixel and the lens is either impacts earlier or later than expected.

Figure 28: Runway cupping

The runway is still quite flatter than the surrounding area as it is situated on top of a hill. One of the more interesting error propagations comes from the lines → width equation. In **Figure 27** shown on page 61 the pitch ±.5° creates a simple error where the lines are still nearly equivalent to the runway lines, just offset by different amounts. In the altitude

from lines error, the equation is more heavily influenced by the ratio of cosine to sine. With hand selected pixels on the white lines, knowing the white line width, the altitude error is shown below in **Figure 29**.

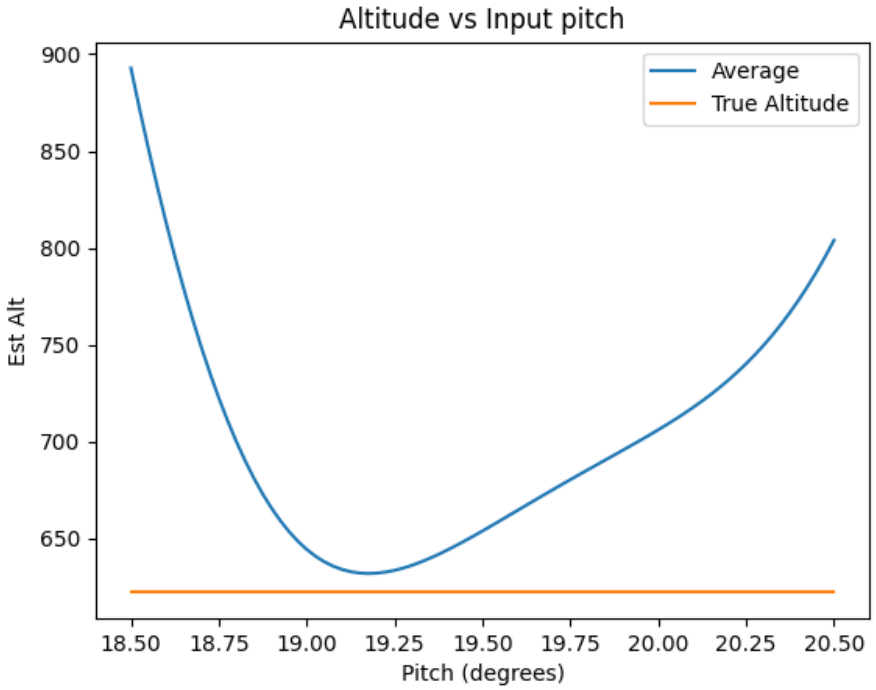

Figure 29: Highly irregular altitude response from slope estimation

The estimated altitude equation never reaches the true altitude for that image and the equation is highly irregular. The minimum error is about 11mm (1.7%) at a pitch of 19.2. The average altitude for values between the 19.5±.5° is 659.37mm for an error of 37.07mm (6%) for this image. In a different test image, where the altitude is 3403mm and pitch is 17.2 ±.5°. The error altitude calculation is 3334.88 ±. 249mm.

Figure 30: Uncorrected drone roll

A very big issue is the actual value of the pitch for these equations. The flight data and the images are separate, with the image creation time being used to figure out the pitch from the flight data recording. From viewing the data corresponding to an image, it appears that the gimbal pitch/roll is not immediately corrected for. In **Figure 30,** the drone camera certainly has a roll component, but it should not exist from the gimbal settings. This could be due to the lag from the drone maneuvering to the gimbal correction. The data polls every 100ms as well, so the information used for the pitch values may be wrong. In a single timestep, the drone moved more than 12° without a change of reading in the gimbal.

This effect extends through to all other estimations since they are based on pitch. All values for the lines in the previous image are generated using the real → pixel values where the x value is +235in, and -255in. And Y value ranges from 0 to the end of the runway. The end of the runway is based on the number of yellow lines and gaps, as well

as the end gap. A gap is 24”, a yellow line is 48” and the end gap is 32”. The ±.5° was from the initial images to capture the errors in the pitch reading and the errors due to the edge of the runway.

The issue is with how the data was gathered. Due to the need of removing any error from GPS and barometers, a method of setting the drone at a given altitude separate was attempted. Without a gantry or other precise methods of recording the runway, a rope tied to the drone with measurements was used. The data recording day was windy as well, causing the rope to catch the wind. The drone needed to not only correct for the impact of the wind on itself, but also the wind on the rope (even if the ground end is held steady, it still impacts the drone).

9. Results

9.1 Overview

The results below are from two different test runs on validation videos, which is video that the drone has never trained on. One test is from a drone recorded altitude of 140ft and the second is from a drone recorded altitude of 20ft. The first example for 140ft has both the corner method and slope method altitudes shown. Only the slope method for 20ft was run, as the corner method was prone to a lot of error.

9.2 Example 1: Altitude of 140ft

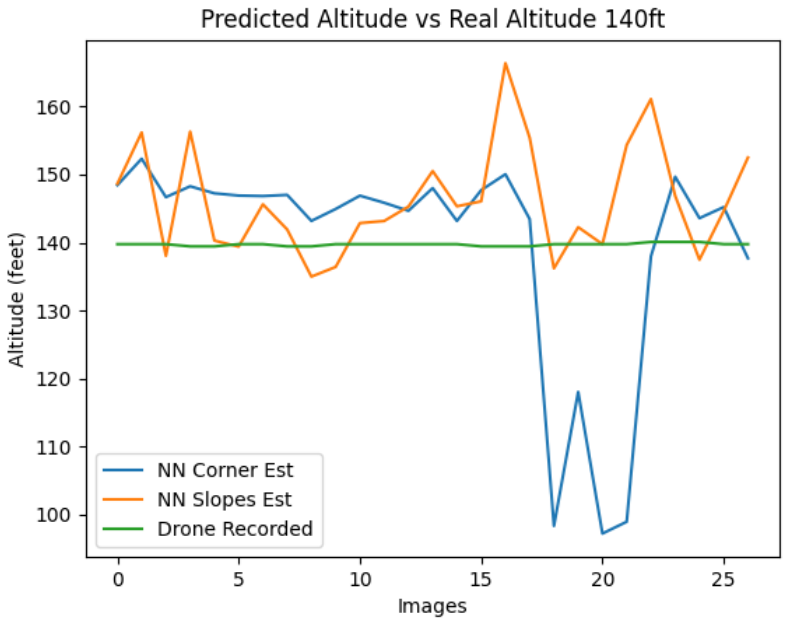

Figure 31: Altitude prediction

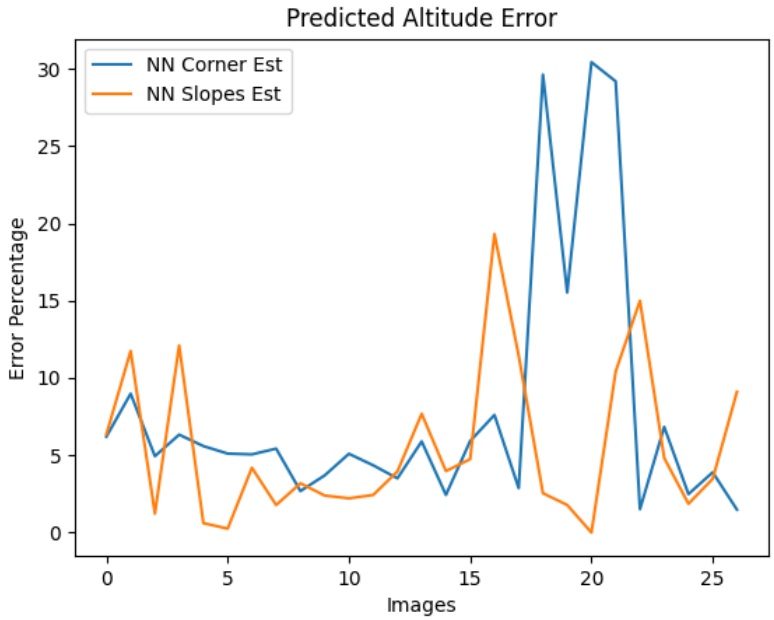

Figure 32: Error in predictions

The large spikes in prediction from frames 15-23 were from far away images causing the edges of the runway to blur in the prediction. Poor runway prediction lines lead to poor corner values from the Ramer–Douglas–Peucker algorithm. In some cases, the slope is more accurate since it is based on four points instead of two, getting a better estimation of the runway. Sometimes the corner estimation is better since it is based on the two corners closest to the camera which are generally better than the further corners. The worst-case error image for the corner method is shown in **Figure 33.** The worst case for the slope estimation error is shown in **Figure 34.**

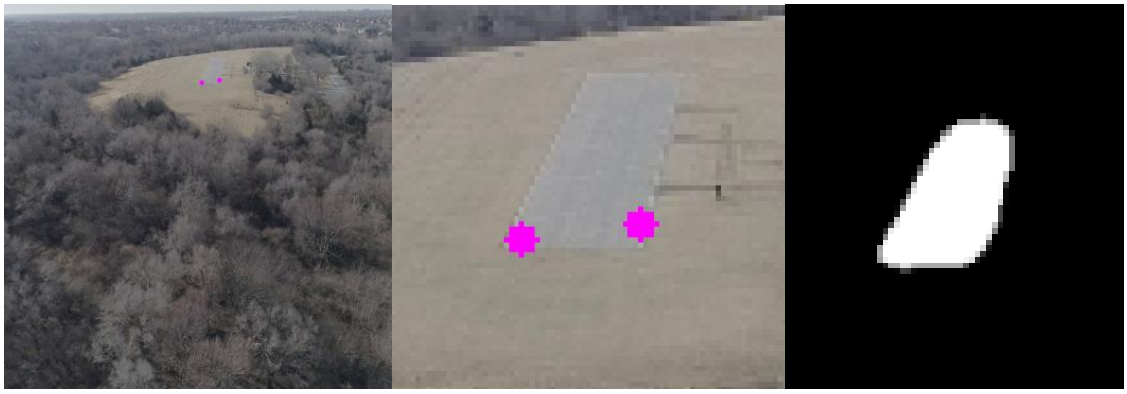

Figure 33: Image with highest error using corner estimation (30.43%) for 140ft (left). Zoomed in image (middle). Zoomed in predicted mask (right)

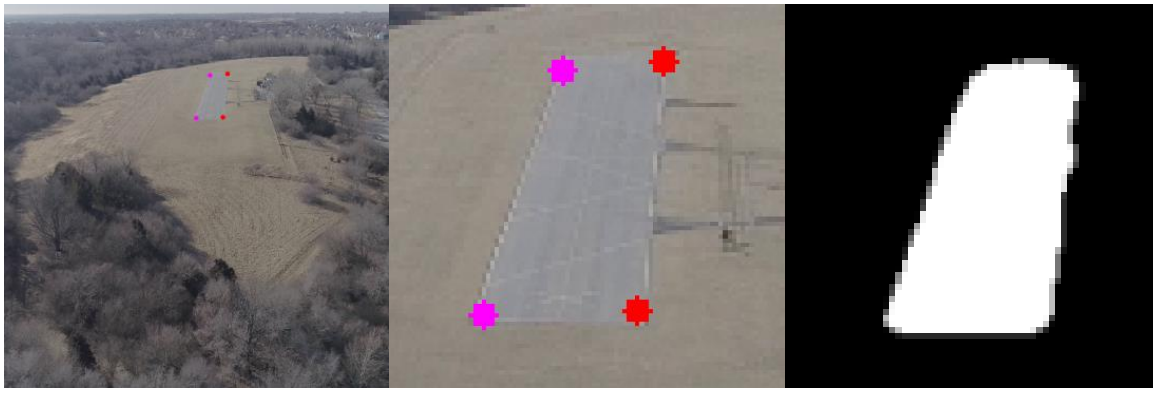

Figure 34: Image with highest error using slope estimation (19.03%) for 140ft (left). Lines used for slopes have the same color end points. Zoomed in image (middle). Zoomed in predicted mask (right)

The neural network time for each image was on average 0.0682s, with a post prediction time average of .0119s for a total average time for altitude prediction of .0801s. This leads to an operation time of 12.48 frames per second. This time is quite variable depending on the altitude of the drone, as seen in the next chapter where the drone is only 20ft above the runway.

9.3 Example 2: Altitude of 20ft

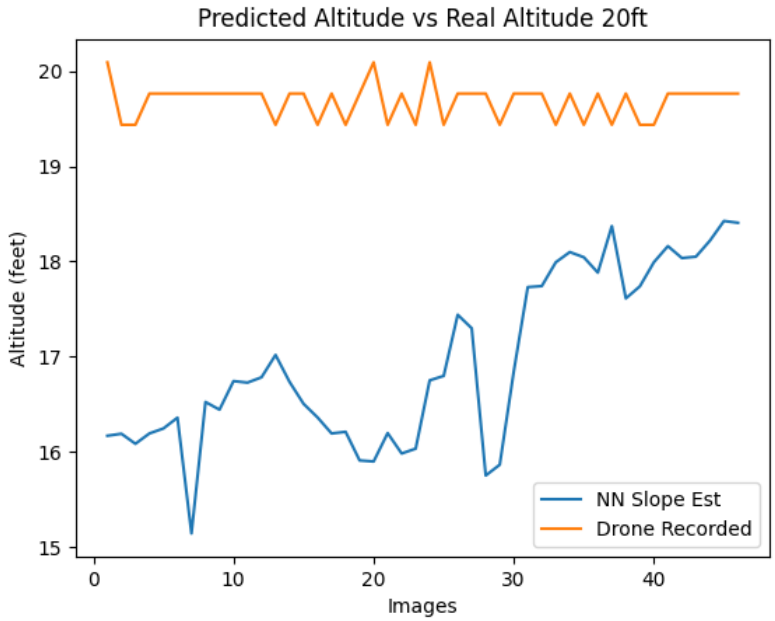

Figure 35: Altitude prediction 20ft

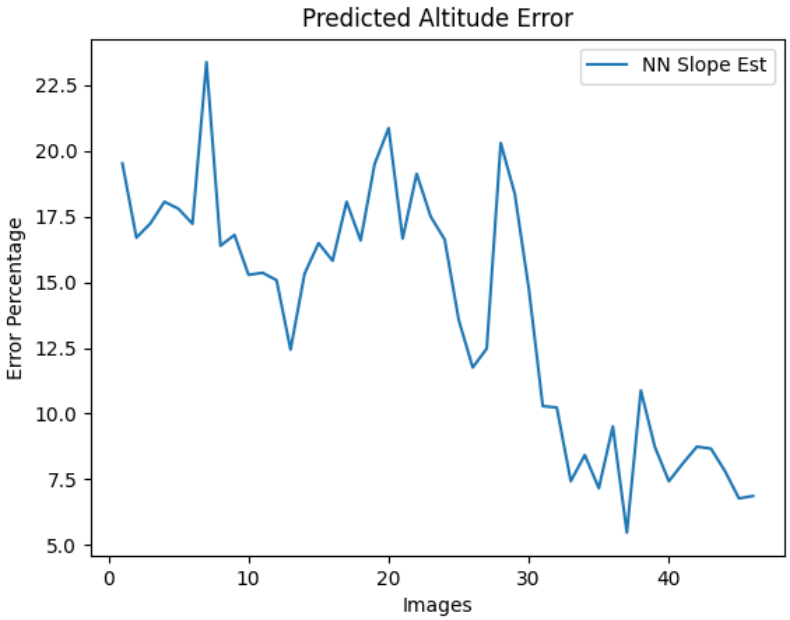

Figure 36: Error in predictions 20ft

An interesting aspect of the data recording is shown in the real altitude for the drone as seen in **Figure 35**. The granularity of the predicted altitude at this point is not small. The altitude is recorded as a multiple of .328084ft or 100mm. The granularity of the altitude is 10 centimeters. The error is on average lower than the 140ft run in distance, but larger in percentage. The corner estimation method had large issues when tested on the 20ft run. As seen in **Figure 37,** the corners must further away since the close corners are not in the image. The neural network was a bit wrong in the prediction, but a sizable portion of the error is also in the polygon approximation. The neural network mask shown in the right image of **Figure 37** and the center image of the zoomed in corners are comprised of the same pixels. The error in altitude for the image in **Figure 37** is 67.48%.

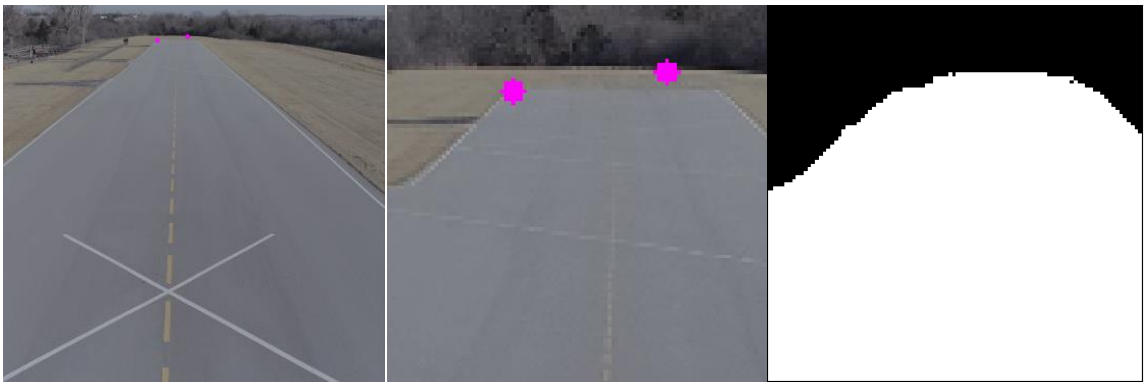

Figure 37: Example image from the 20ft video (left). Zoomed into poor corner prediction (middle). Zoomed in neural network mask (right).

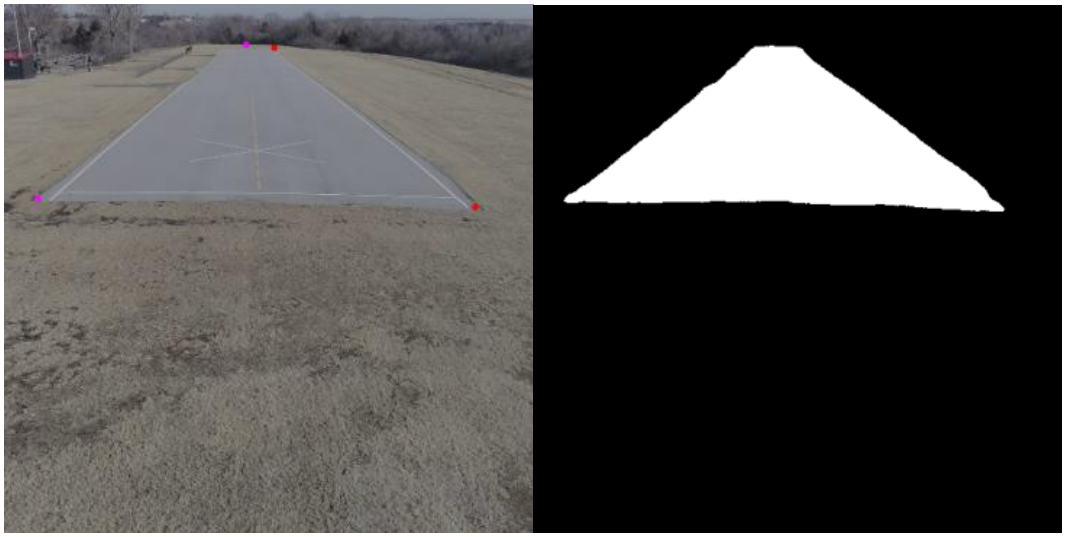

Figure 37. Image with highest error using slope estimation (23.38%) for 20ft (left). Lines used for slopes have the same color end points. Predicted mask (right)

The mask is quite accurate for this image, but the method of polygon approximation recursively removes points based on the distance to a line. This might remove any small feature the image had and destroy the corners. The mask in **Figure 38** is highly accurate, but the top left corner in pink shows how the recovery of the runway shape can cause issues. There might be a better method of getting the corner points of a quadrilateral, but only the Ramer–Douglas–Peucker algorithm was used.

The operation speed was also slower when compared to the 140ft example. Since the runway takes up more of the image, there are more edge points to run through. The neural network time for each image was on average 0.071s, with a post prediction time average of .0157s for a total average time for altitude prediction of .0801s. This leads to an operation time of 11.52 frames per second.

10. Summary

Inaccuracies in GPS and altitude estimations are one of the limiting factors presently in the autonomous landing of drones. Many solutions for vertical landing drones are simple with some defining features for a camera to track. Fixed-wing planes do not have this luxury. There are current methods of landing fixed-wing planes, like ArduPilot, that do not consider the ground features of the landing point. The autopilot can either land at the start location or a user-designated waypoint. This method of landing can work for some drones, but still runs the risk of damage due to the inaccuracy of the altitude estimation and other factors. An additional source of accurate altitude prediction separate from these methods may be able to reduce the error.

The visual light spectrum has been used before as a correction to GPS by fitting a model of the runway to the image [Jbara et. al., 2015]. The assumptions for this method were strict: the drone is currently over the runway and within 10° of the same heading. The commercial airliner Airbus has also completed and evaluated their Autonomous Taxi, Takeoff, and Landing system [Duvelleroy and Benquet, 2020]. Very little information is available for this method, outside of a short article and a picture. Without contact from Airbus, guesses at the method can be made. A neural network method for finding runway location using segmentation on satellite images was tested by Mosaic Data Science [“Detecting Airport Layouts with Computer Vision”, 2021] with decent success. Explored in this paper was a method of runway location and altitude estimation using a neural network segment images with lens equations to reconstruct the distances from the aircraft to the runway.

First, the images for training and targeting are needed to test and train the neural network. These images for this research were recorded during two different seasons at the same airport, Hawk Field, Omaha, NE. An interface to segment the runway in the image was developed based on the placement of four corners either on or off the image. With the four corners describing the runway in an image, the segmentation of the runway is possible. If a point is inside a regular quadrilateral, then the area of the four triangles created by that point and the quadrilateral corners is equal to the area of the quadrilateral, otherwise the area of the triangles is greater. These areas are calculated using the Shoelace theorem. Shoelace theorem requires the points to be in clockwise order, so the points are ordered according to the angle between the vectors to the points.

With the labeled runway segmentation in the image, the neural networks to train on those segmentations was explored. The two primary architectures researched were U-Net and YOLOv3, with many different versions of U-Net evaluated. From the experiments, the best model of U-Net for the computer hardware was determined to be the slightly modified U-Net with 16 initial filters. This model had the highest intersect over union accuracy on the validation data out of all models (.9550), while still operating at a respectable speed (>10FPS). The required post prediction operations of YOLOv3 were not explored due to the additional error of turning the YOLOv3 images into altitude. YOLOv3 only predicts a bounding box, which includes the runway and a significant amount of chaff. However, YOLOv3 had a very high accuracy rate on the validation

images with a significantly faster prediction time. If the goal were only to find if the runway was in the image, YOLOv3 would be preferred.

With the U-Net probability mask, the next step was to determine a threshold for the best predictions. As shown, the standard 0.5 prediction threshold is not always best, so a value of 0.6 was used to reduce the false positives in this research. With the binary prediction image, a chain contouring method was used to determine the edge points on the contours contained in the prediction. The largest contour by area should always be the runway. With all the edge pixels of the runway, the Ramer–Douglas–Peucker algorithm [Douglas and Peucker, 1973] was used to reduce the number of points describing the runway, until only four values remained.

With the corners of the runway found in an image, the next step is to figure out where these points are mapped to the real world. A method of transferring from the real world to the image and from the image back to real world was explored. A homogenous coordinate system to transfer the in-image location to the real-world location was developed based on the pitch and altitude. A few assumptions were made to reduce the complexity of the equations: the real world is a flat plane and the only rotation in the image is pitch. The equations as laid out in this research, are still in a simple, not matrix form like similar methods of coordinate transformation. The reason these equations are kept in a simple format was for the next step.

The ultimate step was predicting the altitude of the drone based on a known dimension on the runway. In this case it was the distance between the concrete edges. The altitude was calculated in two different ways, one using two free floating corners of the runway and the other using four points along the runway edge. This computational method was tested on the experiment images.

11. Contributions and Novel Developments

The strongest contribution shown is the strength of different neural networks at predicting simple runway shapes. The level of accuracy and discrimination is high even when there are difficult images and additional possibly misleading objects in the image. A U-Net model with 16 initial filters was able to perform a prediction at speeds of 12 frames per second or more, while keeping high accuracy.

The modifiable homogenous coordinate system based on the intersection of the real plane and the camera center is novel, as the circumstances that create it are quite unique. The ability to represent the real world as a plane simplifies many of the equations, since the location where the rays generated by the pixel intercept this plane is the location in the real-world plane. The altitude equations based on this coordinate system are also new. If a stable camera was used or more accurate gimbal and pitch information was recorded, the error may fall away leaving an additional accurate method of altitude calculation

The combination of these two methods does leave something to be desired, as the altitude equations are highly sensitive to the small errors of the neural network. The ultimate goal of this system being more accurate than other methods of information gathering was not realized so far in this research.

12. Future Work

As with all neural networks, more data is needed for training the system. Much of the future work would be through the increase of the labeled data, both in amount and diversity. Due to the heavy restrictions placed on drones in the local area, the only good video was from as the singular model runway, Hawk Field. The effect of this was the lack of strong information on how these models work on different and more complex runways. This restriction on drones is throughout the city of Omaha, Nebraska, placed on every park that receives public funding, which is every paved model runway within 3 hours of driving. No other runway was as high quality or as representative of a potential commercial drone runway.

Additionally, the video was taken during two separate periods of time with near the same sunlight, so the color of the runway was very similar between the two recording sessions even if the surrounding color changed. A strong dataset for commercial ventures would need to be expanded to include all possible weather and sunlight conditions. The drone used, the DJI Phantom 4 Pro is not rated for rain or snow, so video during those weather conditions was not possible. Additionally, Hawk Field is surrounded by a residential area, so the primary focus is safety of those residents. Adverse conditions to flying, such as fog, were not recorded.

With this additional data, the best model for prediction might change. Part of the reason the models were able to be reduced to an eighth of the original filter count is because of the homogeneity of the data. If the labeled data was extended to cover more diverse

runways and more diverse weather conditions, more complexity in the models might be needed. However, the research done in this paper should provide a good baseline for comparison.

The exact position and location information for testing the data was also limited. Since the altitude as recorded by the drone is approximate, a different system for determining exact altitude is needed. In this paper, that method was still very error prone. Other methods of recording the altitude with less error, would be beneficial for any future work.

Separate from more underlying data, the equations used for the homogenous coordinate system can be flushed out to include all degrees of freedom for the camera. If the drone rotation data and gimbal rotation data are known, the homogenous system can be created. The equations become incredibly more complex for a doubly rotated plane and matrix. However, these rotation equations are not new and can be implemented into the system outlined in this paper since it is simply based on ray intersections.

Accuracy can also be improved for all aspects of this project if the camera on the drone and the drone inertial measurement unit were less erroneous. Since the camera pitch angle includes both error from the inertial measurement unit and the gimbal system, the outcome of the post image operations contains high levels of error.

The runway location and altitude values calculated from the images can also be fed into the other methods of estimation to provide better quality data. A Kalman Filter attempts

to reduce the error in many diverse sources for the same information. The filter in ArduPilot attempts to reduce the error in altitude between GPS and barometric pressure. A next step may be to include the estimated altitude from this paper into the Kalman Filter. This may be of limited use since the Kalman filter was developed for statistical error, but the error in neural networks and the information gleaned from them is nowhere near a normal distribution.

References

Abraham, N., and Khan, N.M. "A Novel Focal Tversky Loss Function with Improved Attention U-Net for Lesion Segmentation", *2019 IEEE 16th International Symposium on Biomedical Imaging*, 2019, pp. 683-687

Abu-Jbara, K., Alheadary. W., Sundaramorthi, G., and Claudel, C., "A robust vision-based runway detection and tracking algorithm for automatic UAV landing", *Proceedings of 2015 International Conference on Unmanned Aircraft Systems (ICUAS)*, 2015, pp. 1148-1157

Albawi, S., Mohammed, T. A., Al-Zawi, S., "Understanding of a convolutional neural network", *Proceedings of 2017 International Conference on Engineering and Technology (ICET)*, 2017, pp. 1-6

Beheshti N. and Johnsson L., "Squeeze U-Net: A Memory and Energy Efficient Image Segmentation Network," *IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops*, 2020, pp. 1495-1504

Braden B., "The Surveyor's Area Formula," *The College Mathematics Journal*, vol. 17, no. 4, 1986, pp. 326-337

Brownlee, J., "How to Perform Object Detection with YOLOv3 in Keras", https://machinelearningmastery.com/how-to-perform-object-detection-with-yolov3-in-keras, May 2019

"Detecting Airport Layouts with Computer Vision", https://mosaicdatascience.com/wp-content/uploads/2021/07/mds-detecting-airport-layouts-with-computer-vision-white-paper.pdf, July 2021

Ding, X., Zhang, X., Zhou, Y., Han, J., Ding, G., Sun, J., "Scaling Up Your Kernels to 31x31: Revisiting Large Kernel Design in CNNs", https://arxiv.org/pdf/2203.06717.pdf, Apr. 2022

Douglas, D.H., and Peucker, T.K., "Algorithms for the Reduction of the Number of Points Required to Represent a Digitized Line or its Caricature.," *Cartographica: The International Journal for Geographic Information and Geovisualization*, Volume 10, 1973, pp. 112-122.

Dozat, Timothy. "Incorporating Nesterov Momentum into Adam.", 2016

Duthon, P., Colomb M., Bernardin F., "Light Transmission in Fog: The Influence of Wavelength on the Extinction Coefficient" Applied Sciences 9, no. 14: 2843, 2019

Duvelleroy, M., Benquet L., “Airbus concludes ATTOL with fully autonomous flight tests” https://www.airbus.com/en/newsroom/press-releases/2020-06-airbus-concludes-attol-with-fully-autonomous-flight-tests, June 2020

Federal Aviation Administration, “ENR 1.7 Barometric Altimeter Errors and Setting Procedures”. *General Rules and Procedures*. https://www.faa.gov/air_traffic/publications/atpubs/aip_html/part2_enr_section_1.7.html, as of July 2022

Felzenszwalb, P. F., Girshick, R. B, McAllester, D., Ramanan, D., "Object Detection with Discriminatively Trained Part-Based Models," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 32, no. 9, Sept. 2010, pp. 1627-1645

Fukushima, K. “Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position.”, *Biological Cybernetics,* Volume 36, 1980, pp. 193–202.

Gallagher, Eugene. “COMPAH Documentation”, 2004, pp. 12-16

“GPS Accuracy”, https://www.gps.gov/systems/gps/performance/accuracy/#:~:text=Learn%20more-How%20accurate%20is%20GPS%20for%20speed%20measurement%3F,interval%2C%20with%2095%25%20probability, as of July 2022

Hartley, R., Zisserman, A., “Multiple View Geometry”, Cambridge University Press, ISBN 978-0-521-54051-3, 2004

He K., Zhang X., Ren, S., and Sun, J., "Deep Residual Learning for Image Recognition,” IEEE *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 770-778

“How is Altitude Measured in Aviation”, https://hartzellprop.com/altitude-measured-aviation/, May 2018

Jadon S., "A survey of loss functions for semantic segmentation," *IEEE Conference on Computational Intelligence in Bioinformatics and Computational Biology (CIBCB)*, 2020, pp. 1-7

Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R., Guadarrama, S., Darrell, T., “Caffe: Convolutional Architecture for Fast Feature Embedding” https://arxiv.org/pdf/1408.5093.pdf, June 2014

Kalman, R.E., “A New Approach to Linear Filtering and Prediction Problems”, *Transactions of the ASME--Journal of Basic Engineering,* Volume 82, 1959.pp. 35-45

Kim, S., Woo, R., Yang, E., Seo, D., Real Time Multi-Lane Detection Using Relevant Lines Based on Line Lableling Method", *Proceedings of The 4th International Conference on Intelligent Transportation Engineering*, Sept 2019

Labun, J., Kurdel, P., Ceškovič, M., Nekrasov, A., Gamec, J., "Low Altitude Measurement Accuracy Improvement of the Airborne FMCW Radio Altimeters", Multidisciplinary Digital Publishing Institute, June 2019

Lukács L., "In-Flight Horizon Line Detection for Airplanes Using Image Processing", *Proceedings of IEEE 13th International Symposium on Intelligent Systems and Informatics*, 2015

Mili, "How to determine if a point is within a quadrilateral,". *https://stackoverflow.com/questions/5922027/how-to-determine-if-a-point-is-within-a-quadrilateral*, April 2013, as of July 2022

Pavithran, P., "How to label custom images for YOLO – YOLO 3", https://cloudxlab.com/blog/label-custom-images-for-yolo/, as of July 2022

Redmon, J., Divvala, S., Girshick R., Farhadi, A., "You Only Look Once: Unified, Real-Time Object Detection," *Proceedings of 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2016, pp. 779-788*

Redmon, J. and Farhadi, A., "YOLOv3: An Incremental Improvement". https://arxiv.org/pdf/1804.02767.pdf, Apr 2018

Ronneberger, O., Fischer, P., and Brox T., "U-Net: Convolutional Networks for Biomedical Image Segmentation", https://arxiv.org/pdf/1505.04597.pdf, May 2015

Simonyan, K., & Zisserman, A., "Very Deep Convolutional Networks for Large-Scale Image Recognition.", https://arxiv.org/abs/1409.1556, Apr 2015

Technical University of Munich, "Successful Automatic Landing with Vision Assisted Navigation" https://www.tum.de/en/about-tum/news/press-releases/details/35556/, Apr 2019

Wolkow, S., Angermann, M., Dekiert, A., Bestmann, U., "Model-based Threshold and Centerline Detection for Aircraft Positioning during Landing Approach," *Proceedings of the ION 2019 Pacific PNT Meeting*, Honolulu, Hawaii, April 2019, pp. 767-776.

Zhao, K., Liu, Y., Hao, S., Lu S., Liu, H. Zhou, L., "Bounding Boxes Are All We Need: Street View Image Classification via Context Encoding of Detected Buildings," IEEE *Transactions on Geoscience and Remote Sensing*, vol. 60, 2022, pp. 1-17

Appendix A Target Generating User Interface

All code will be slowly added to this repository:
https://github.com/jludeman-1/Ludeman-Thesis-Code/

The code may be different than shown in this thesis, as I will continue to work on this in my free time.

```
"""
UI for target generation.
Based on 4 corners inside or outside of image
Toggle Bounds button allows for the corner to be on the edge of image
without going outside
    - A move while bounds are on will set an out of bounds corner to
the bound of the movement direction

Data path is where to store the generated CSV file of the corners
Image path is the path of the images-1920x1080
"""


import csv
from PyQt5.QtGui import QPixmap, QImage

from PyQt5.QtWidgets import QApplication, QMainWindow, QLabel,
QPushButton, \
    QButtonGroup, QRadioButton
from PyQt5 import QtCore

import sys
import cv2
from os import listdir
from utils import box



class MainWindow(QMainWindow):
    def __init__(self, parent=None):
        super(MainWindow, self).__init__(parent)

        self.dataPath = "E:\\Thesis\\code\\data\\video13\\"
        self.imagePath =
"E:\\Thesis\\code\\data\\images\\full\\video2\\"

        self.setWindowTitle("Targeting")

        # Can set count if targeting starts from a different image
count
        # Be sure to save current CSV since it is overwritten

        self.count = 0

        # Scale of image in UI
```

```
        self.scale = .9


        self.move(0,0)

        # Image names from images in imagePath
        self.frameIDX = sorted([int(f[5:(len(f) - 4)]) for f in
listdir(self.imagePath)])
        # self.count = self.frameIDX.index(620)

        self.imgCount = len(self.frameIDX)

        # Starting image name
        self.frameName = "frame%d.png" % self.frameIDX[self.count]

        self.img = cv2.imread(self.imagePath + self.frameName)

        # Image shown in UI
        self.imgLines = self.img.copy()

        # Box is found in Utils
        self.target = box([self.img.shape[0], self.img.shape[1]])
        self.target.draw(self.imgLines, thickness=1)


        self.baseDims = self.img.shape
        width = int(self.img.shape[1] * self.scale)
        height = int(self.img.shape[0] * self.scale)
        self.scaledDim = (width, height)

        # Interpolation method tries to preserve lines in scaled images
        self.imgScaled = cv2.resize(self.imgLines, self.scaledDim,
interpolation=cv2.INTER_AREA)

        self.image = QImage(self.imgScaled.data,
self.imgScaled.shape[1],
self.imgScaled.shape[0],QImage.Format_RGB888).rgbSwapped()
        self.label = QLabel(self)
        self.label.setPixmap(QPixmap.fromImage(self.image))


        # all the following lines are setting up buttons
        # No focus set for each button necessary for arrow key
integration
        self.label.resize(self.image.width(), self.image.height())
        self.resize(self.image.width(), self.image.height())

        self.countLabel = QLabel(self)
        self.countLabel = QLabel(str(self.count), self)
        self.countLabel.move(110, 1)

        self.countLabel2 = QLabel(self)
        self.countLabel2 = QLabel('/' + str(len(self.frameIDX)), self)
        self.countLabel2.move(130, 1)


        self.resize(self.width() + 170, self.image.height() + 20)
```

```
self.label.move(170, 0)
self.lineLabel = QLabel('Corner', self)
self.lineLabel.move(1, 0)

self.linegroup = QButtonGroup(self)

baseForLabels = 7
labelOffset = 20
buttonHeight = 40
line1Btn = QRadioButton("1", self)
line1Btn.move(5, buttonHeight)
line1Btn.resize(15, line1Btn.height())
line1Label = QLabel('1', self)
line1Label.move(baseForLabels, 20)
line1Btn.setFocusPolicy(QtCore.Qt.NoFocus)
line2Btn = QRadioButton("2", self)
line2Btn.move(25, buttonHeight)
line2Btn.resize(15, line1Btn.height())
line2Label = QLabel('2', self)
line2Label.move(baseForLabels + labelOffset, 20)
line2Btn.setFocusPolicy(QtCore.Qt.NoFocus)
line3Btn = QRadioButton("3", self)
line3Btn.move(45, buttonHeight)
line3Btn.resize(15, line1Btn.height())
line3Label = QLabel('3', self)
line3Label.move(baseForLabels + labelOffset * 2, 20)
line3Btn.setFocusPolicy(QtCore.Qt.NoFocus)
line4Btn = QRadioButton("4", self)
line4Btn.move(65, buttonHeight)
line4Btn.resize(15, line1Btn.height())
line4Label = QLabel('4', self)
line4Label.move(baseForLabels + labelOffset * 3, 20)
line4Btn.setFocusPolicy(QtCore.Qt.NoFocus)
line1Btn.setChecked(True)

self.linegroup.addButton(line1Btn, 1)
self.linegroup.addButton(line2Btn, 2)
self.linegroup.addButton(line3Btn, 3)
self.linegroup.addButton(line4Btn, 4)
# self.linegroup.
cornerLabel = QLabel('Move Corner Horizontal', self)
cornerLabel.move(5, 60)


hNegOneBtn = QPushButton("+1", self)
hNegOneBtn.setFocusPolicy(QtCore.Qt.NoFocus)
hNegFiveBtn = QPushButton("+5", self)
hNegFiveBtn.setFocusPolicy(QtCore.Qt.NoFocus)
hNegTwentyFiveBtn = QPushButton("+25", self)
hNegTwentyFiveBtn.setFocusPolicy(QtCore.Qt.NoFocus)
hNegHundredBtn = QPushButton("+100", self)
hNegHundredBtn.setFocusPolicy(QtCore.Qt.NoFocus)
hNegOneBtn.move(5, 90)
hNegOneBtn.resize(35, 20)

hNegFiveBtn.move(45, 90)
hNegFiveBtn.resize(35, 20)
```

```
        hNegTwentyFiveBtn.move(85, 90)
        hNegTwentyFiveBtn.resize(35, 20)

        hNegHundredBtn.move(125, 90)
        hNegHundredBtn.resize(35, 20)

        hPosOneBtn = QPushButton("-1", self)
        hPosFiveBtn = QPushButton("-5", self)
        hPosTwentyFiveBtn = QPushButton("-25", self)
        hPosHundredBtn = QPushButton("-100", self)

        hPosOneBtn.setFocusPolicy(QtCore.Qt.NoFocus)
        hPosFiveBtn.setFocusPolicy(QtCore.Qt.NoFocus)
        hPosTwentyFiveBtn.setFocusPolicy(QtCore.Qt.NoFocus)
        hPosHundredBtn.setFocusPolicy(QtCore.Qt.NoFocus)

        hPosOneBtn.move(5, 110)
        hPosOneBtn.resize(35, 20)


if __name__ == "__main__":
    sys._excepthook = sys.excepthook


    def exception_hook(exctype, value, traceback):
        print(exctype, value, traceback)
        sys._excepthook(exctype, value, traceback)
        sys.exit(1)


    sys.excepthook = exception_hook
    app = QApplication(sys.argv)
    window = MainWindow()
    app.exec_()
```

Appendix B Utilities

```
import cv2
import numpy
from ctypes import CDLL, c_int, POINTER, c_double
import numpy as np

# C shared object for TriArea shared object
so_file = "./cCode/TriArea.so"
func = CDLL(so_file)

func.TriArea.restype = c_double
func.testPoint.restype = c_double
func.generateMaskPadded.restype = c_double

# C shared object for preprocessing shared object
so_file2 = "./cCode/preprocessing.so"
preprocessing = CDLL(so_file2)
preprocessing.threshold3Dto3D.restype = c_double


# Applies threshold to the given input prediction
# INPUTS
#   predIn Prediction mask in pixel probability
#   thresh Value to threshold the prediction by
def applyThresh(predIn, thresh):
    for row in range(512):
        for col in range(512):
            if predIn[row, col] > thresh:
                predIn[row, col] = 1
            else:
                predIn[row, col] = 0
    return predIn


# Returns iou, dice, sens, spec for input threshold and .5
# INPUTS
#   maskIn  True binary mask
#   predIn Prediction mask in pixel probability
#   threshold Value to threshold the prediction by
def fourMetrics(maskIn, predIn, threshold):
    TP = 0
    FP = 0
    TN = 0
    FN = 0

    TP5 = 0
```

```
    FP5 = 0
    TN5 = 0
    FN5 = 0

    for row in range(512):
        for col in range(512):

            if predIn[row, col] > threshold:
                predVal = 1
            else:
                predVal = 0

            if predVal == maskIn[row, col] and predVal == 1:

                TP += 1
            elif predVal == maskIn[row, col] and predVal == 0:

                TN += 1
            elif predVal != maskIn[row, col] and predVal == 0:

                FN += 1
            elif predVal != maskIn[row, col] and predVal == 1:

                FP += 1

            if predIn[row, col] >= .5:
                predVal5 = 1
            else:
                predVal5 = 0

            if predVal5 == maskIn[row, col] and predVal5 == 1:

                TP5 += 1
            elif predVal5 == maskIn[row, col] and predVal5 == 0:

                TN5 += 1
            elif predVal5 != maskIn[row, col] and predVal5 == 0:

                FN5 += 1
            elif predVal5 != maskIn[row, col] and predVal5 == 1:

                FP5 += 1

    # Catches div/0 error when there are no true positives and the NN
is completely correct
    # IE an image has no runway and the model doesnt predict any runway
pixels
    # IoU is then based on rate of true negatives
    if not TP == 0:
        iou = TP / (TP + FP + FN)

    else:
        iou = (TP + TN) / (TP + TN + FP + FN)

    if not TP5 == 0:
        iou5 = TP5 / (TP5 + FP5 + FN5)
```

```
    else:
        iou5 = (TP5 + TN5) / (TP5 + TN5 + FP5 + FN5)
    dice = 2 * iou / (1 + iou)
    sens = (TP + 1) / (TP + FN + 1)
    spec = (TN + 1) / (TN + FP + 1)

    dice5 = 2 * iou5 / (1 + iou)
    sens5 = (TP5 + 1) / (TP5 + FN5 + 1)
    spec5 = (TN5 + 1) / (TN5 + FP5 + 1)
    return iou, dice, sens, spec, iou5, dice5, sens5, spec5


# For a given label and prediction, calculate the IOU values for
thresholds .1-.10
# INPUTS
#   maskIn 512x512x1 numpy array
#   predIn 512x512x1 numpy array
#   iouArray array to store values in
def calcThreshold(maskIn, predIn, iouArray):
    for thresh in range(10):

        TP = 0
        FP = 0
        TN = 0
        FN = 0
        threshold = thresh / 10

        for row in range(512):
            for col in range(512):

                if predIn[row, col] > threshold:
                    predVal = 1
                else:
                    predVal = 0

                if predVal == maskIn[row, col] and predVal == 1:

                    TP += 1
                elif predVal == maskIn[row, col] and predVal == 0:

                    TN += 1
                elif predVal != maskIn[row, col] and predVal == 0:

                    FN += 1
                elif predVal != maskIn[row, col] and predVal == 1:

                    FP += 1
        if not (TP + FP + FN) == 0:
            iouArray[thresh] = TP / (TP + FP + FN)
        else:
            iouArray[thresh] = 0
    print("%.1f,   %.5f,   %.5f" % (numpy.argmax(iouArray) / 10,
iouArray[numpy.argmax(iouArray)], numpy.max(predIn)))


# Comparison of Points based on the cross product of the two vectors
# Based on Code provided by
```

```
https://stackoverflow.com/questions/6989100/sort-points-in-clockwise-
order
# INPUTS
#   p1 1x2 numpy array x,y or col, row
#   p2 1x2 numpy array x,y or col, row
#   center 1x2 numpy array barycenter x, y or col, row
def shoeLaceComparison(p1, p2, center):
    # Cross Product of the two vectors
    # The equation is AXB =||A|| * ||B|| * sin(angle)
    # If negative, the angle between A and B is negative
    cross = (p1[1] - center[1]) * (p2[0] - center[0]) - (p2[1] -
center[1]) * (p1[0] - center[0])

    # If negative, the angle between A and B is negative
    if cross < 0:
        return True

    # If Positive, the angle between A and B is positive
    if cross > 0:
        return False

    # Distance from barycenter (square root not done, only servers as
added complexity)
    d1 = (p1[1] - center[1]) * (p1[1] - center[1]) + (p1[0] -
center[0]) * (p1[0] - center[0])
    d2 = (p2[1] - center[1]) * (p2[1] - center[1]) + (p2[0] -
center[0]) * (p2[0] - center[0])

    return d1 > d2


# Class for storing the corners of the target set by a person
# Input are the base limits for the corners determined by the base
image
# Corners are stored in the order [row,column]
# Be aware, most cv2 methods are in order [horizontal, vertical] or
[column, row]
class box:
    def __init__(self, baseMax):
        self.maxVals = baseMax
        self.corners = numpy.array([[5, 5], [50, 5], [5, 100], [50,
100]]).astype(int)
        c_float_p = POINTER(c_int)
        self.cornerPointer = self.corners.ctypes.data_as(c_float_p)

        # Generating target image is quite intensive, only done once if
possible
        self.targetGenerated = False
        self.state = 'n'

    # Moves corner by amount, idx is row or column
    # UI integration, never called by user
    # INPUTS
    #   cornerNum corner idx in list to be moved
    #   amount int amount of corner to be moved
    #   idx int if x or y is changed
    #   cornerBounds boolean if cornerBounds are enforced
```

```
    def moveCorner(self, cornerNum, amount, idx, cornerBounds):
        if self.corners[cornerNum - 1][idx] + amount >=
self.maxVals[idx] and not cornerBounds:
            self.corners[cornerNum - 1][idx] = self.maxVals[idx] - 1
        elif self.corners[cornerNum - 1][idx] + amount <= 0 and not
cornerBounds:
            self.corners[cornerNum - 1][idx] = 0
        else:
            self.corners[cornerNum - 1][idx] += amount

    # Draws box onto input image, copy is whether or not to copy image
    # INPUTS
    #   imgIn where to draw
    #   thickness int thickness of lines
    #   copy return copy of image
    #   binary values are 0/1
    def draw(self, imgIn, thickness=2, copy=False, binary=False):

        self.sortCorners()

        if copy:
            imgOut = imgIn.copy()
        else:
            imgOut = imgIn
        if binary:
            color = [255]
        else:
            color = [222, 222, 38]
        # print(self.corners)
        cv2.line(imgOut, [self.corners[0][1], self.corners[0][0]],
[self.corners[1][1], self.corners[1][0]], color,
                 thickness)
        cv2.line(imgOut, [self.corners[1][1], self.corners[1][0]],
[self.corners[2][1], self.corners[2][0]], color,
                 thickness)
        cv2.line(imgOut, [self.corners[2][1], self.corners[2][0]],
[self.corners[3][1], self.corners[3][0]], color,
                 thickness)
        cv2.line(imgOut, [self.corners[3][1], self.corners[3][0]],
[self.corners[0][1], self.corners[0][0]], color,
                 thickness)
        return imgOut

    # Sets corners
    def set(self, newCorners):
        self.corners = numpy.array(newCorners)

    # Sorts corners based on cross product between the two points
    def sortCorners(self):
        newCorners = self.corners.copy().tolist()
        baryCenter = [int((newCorners[0][0] + newCorners[1][0] +
newCorners[2][0] + newCorners[3][0]) / 4),
                      int((newCorners[0][1] + newCorners[1][1] +
newCorners[2][1] + newCorners[3][1]) / 4)]

        clockwise = False
        idx = 0
```

```
        while not clockwise:

            if not shoeLaceComparison(newCorners[idx], newCorners[idx +
1], baryCenter):
                idx += 1
            else:
                tempC = newCorners[idx]
                newCorners[idx] = newCorners[idx + 1]
                newCorners[idx + 1] = tempC
                idx = 0
            if idx == 3:
                clockwise = True
        self.set(newCorners)

    # Draws corners
    # INPUTS
    #   imgIn where to draw
    def drawCorners(self, imageIn):
        self.sortCorners()
        newCorners = self.corners
        cv2.circle(imageIn, [newCorners[0][1], newCorners[0][0]], 0,
[255, 0, 0], 2)
        cv2.circle(imageIn, [newCorners[1][1], newCorners[1][0]], 0,
[255, 0, 255], 10)
        cv2.circle(imageIn, [newCorners[2][1], newCorners[2][0]], 0,
[255, 255, 255], 20)
        cv2.circle(imageIn, [newCorners[3][1], newCorners[3][0]], 0,
[0, 255, 0], 30)
        return imageIn

    # Fill in target on slice of image
    def fillSlice(self, dimsIn):

        self.sortCorners()
        outMat = numpy.zeros(dimsIn).astype(int)

        out_P = POINTER(c_int)
        outMatPointer = outMat.ctypes.data_as(out_P)

        corners = self.corners.copy()

        c_int_pointer = POINTER(c_int)

        cornerPointer = corners.ctypes.data_as(c_int_pointer)

        func.generateMaskPadded(outMatPointer, cornerPointer,
dimsIn[0], dimsIn[1])

        arr1 = numpy.reshape(outMat, (-1, dimsIn[1]))
        # print(sumArr1)
        return arr1

    # Fill in target on full image
    def fillTarget(self):

        self.sortCorners()
        outMat = numpy.zeros([1080 * 1920]).astype(int)
```

```
        out_P = POINTER(c_int)
        outMatPointer = outMat.ctypes.data_as(out_P)

        # corners = numpy.array([[33, 833], [31, 1215], [1079, 2259],
[1079, -185]])

        c_int_pointer = POINTER(c_int)

        cornerPointer =
self.corners.copy().ctypes.data_as(c_int_pointer)

        func.generateMaskPadded(outMatPointer, cornerPointer, 1080,
1920)

        arr1 = numpy.reshape(outMat, (-1, 1920))
        return arr1


# Class that stores all needed information for target generation
class TargetImage:
    def __init__(self, imageIn, name, basePath='./', yoloPath='./',
unetPath='./'):
        # self.corners = None
        self.imgPath = basePath
        self.yoloPath = yoloPath
        self.unetPath = unetPath
        self.box = box(imageIn.shape)
        self.image = imageIn
        self.imageName = name
        self.targetGenerated = False
        self.targetImg = numpy.zeros([imageIn.shape[0],
imageIn.shape[1], 1])

    def generateTarget(self):
        self.targetImg = self.box.fillTarget()
        self.targetGenerated = True

    def save(self):
        cv2.imwrite(self.imgPath + 'testTarg.png', self.image)

    # Generates the U-Net Targets based on the input dimensions
    #   -ONLY SET UP FOR 512x512 RIGHT NOW
    #       -EDGE OFFSETS NEED TO BE CHANGED FOR NON 512x512
    # INPUTS
    #   dimsIn dimension of slice [512x512]
    #   step int save name step.png
    #   padded boolean is convolution padded?
    def generateUnetTarget(self, dimsIn, step=0, padded=True):
        self.box.sortCorners()

        downScaled3 = cv2.resize(self.image[0:1080, 420:1920 - 420],
dimsIn)

        scaleFactor = dimsIn[0] / 1080

        baseCorners = self.box.corners.copy()
```

```
        self.box.corners = baseCorners.copy()

        self.box.corners[:, 1] = self.box.corners[:, 1] - 420
        self.box.corners = (self.box.corners *
scaleFactor).round().astype(int)
        t3 = self.box.fillSlice(dimsIn)

        self.box.corners = baseCorners.copy()

        if not padded:
            t3 = t3[(0 + 92):(572 - 92), (0 + 92): (572 - 92)]

        cv2.imwrite(self.imgPath + "%d" % step + ".png", downScaled3)
        cv2.imwrite(self.unetPath + "%d" % step + ".png",
                    t3)
        step += 1

        return step

    # Generates the YOLOv3 Targets based on the input dimensions
    #   -ONLY SET UP FOR 512x512 RIGHT NOW
    #       -EDGE OFFSETS NEED TO BE CHANGED FOR NON 512x512
    # INPUTS
    #   dimsIn dimension of slice [512x512]
    #   step int save name step.png
    #   test boolean images will have targets drawn on
    def generateYOLOTarget(self, dimsIn, step=0, test=False):
        self.box.sortCorners()

        downScaled = cv2.resize(self.image[0:1080, 420:1920 - 420],
dimsIn)

        scaleFactor = dimsIn[0] / 1080

        baseCorners = self.box.corners.copy()

        self.box.corners = baseCorners.copy()

        self.box.corners[:, 1] = self.box.corners[:, 1] - 420
        self.box.corners = (self.box.corners *
scaleFactor).round().astype(int)
        unetTarget = self.box.fillSlice(dimsIn)

        reverseIndex = dimsIn[0] - 1
        rowMin = -1
        rowMax = -1
        colMin = -1
        colMax = -1
        for idx in range(dimsIn[0]):

            if rowMin == -1 and not np.sum(unetTarget[idx, :]) == 0:
                rowMin = idx

            if rowMax == -1 and not np.sum(unetTarget[reverseIndex -
idx, :]) == 0:
                rowMax = reverseIndex - idx
```

```
            if colMin == -1 and not np.sum(unetTarget[:, idx]) == 0:
                colMin = idx

            if colMax == -1 and not np.sum(unetTarget[:, reverseIndex -
idx]) == 0:
                colMax = reverseIndex - idx

            if colMax != -1 and colMin != -1 and rowMax != -1 and
rowMin != -1:
                break

        self.box.corners = baseCorners.copy()

        corners = numpy.array([[rowMin, colMin], [rowMin, colMax],
[rowMax, colMax],
                               [rowMax, colMin]])

        if numpy.sum(unetTarget) == 0:
            noCorners = True
        else:
            noCorners = False

        if test:
            if not noCorners:
                color = [222, 222, 38]
                thickness = 1

                cv2.line(downScaled, [corners[0][1], corners[0][0]],
[corners[1][1], corners[1][0]], color,
                         thickness)
                cv2.line(downScaled, [corners[1][1], corners[1][0]],
[corners[2][1], corners[2][0]], color,
                         thickness)
                cv2.line(downScaled, [corners[2][1], corners[2][0]],
[corners[3][1], corners[3][0]], color,
                         thickness)
                cv2.line(downScaled, [corners[3][1], corners[3][0]],
[corners[0][1], corners[0][0]], color,
                         thickness)
                # print("saving to: " + self.imgPath + "%d" % step +
".png", )
                cv2.imwrite(self.imgPath + "%d" % step + ".jpg",
downScaled)
            else:
                print(step)
                cv2.imwrite(self.imgPath + "%d" % step + ".jpg",
downScaled)

        else:
            cv2.imwrite(self.imgPath + "%d" % step + ".jpg",
downScaled)
            if not noCorners:

                xCenter = ((colMax + colMin) / 2) / 416
                yCenter = ((rowMax + rowMin) / 2) / 416
                width = (colMax - colMin) / 416
                height = (rowMax - rowMin) / 416
```

```
                with open(r"" + self.yoloPath + "%d.txt" % step, 'w')
as fp:
                    fp.write("0 %.5f %.5f %.5f %.5f" % (xCenter,
yCenter, width, height))
            else:

                with open(r"" + self.yoloPath + "%d.txt" % step, 'w')
as fp:
                    fp.write("")

        self.box.corners = baseCorners.copy()
```

Appendix C Neural Network Models

```
# Basic U-Net Model
def unet(filters=64, input_size=(512, 512, 3)):
    inputs = Input(input_size)
    l1Conv1 = Conv2D(filters, 3, activation='relu', padding='same',
kernel_initializer='he_normal', name='l1Conv1')(
        inputs)
    spDrop1 = SpatialDropout2D(.2)(l1Conv1)
    l1Conv2 = Conv2D(filters, 3, activation='relu', padding='same',
kernel_initializer='he_normal', name='l1Conv2')(
        spDrop1)
    pool1 = MaxPooling2D(pool_size=(2, 2), name='pool1')(l1Conv2)

    l2Conv1 = Conv2D(filters * 2, 3, activation='relu', padding='same',
kernel_initializer='he_normal', name='l2Conv1')(
        pool1)
    spDrop2 = SpatialDropout2D(.2)(l2Conv1)
    l2Conv2 = Conv2D(filters * 2, 3, activation='relu', padding='same',
kernel_initializer='he_normal', name='l2Conv2')(
        spDrop2)
    pool2 = MaxPooling2D(pool_size=(2, 2), padding='same',
name='pool2')(l2Conv2)

    l3Conv1 = Conv2D(filters * 4, 3, activation='relu', padding='same',
kernel_initializer='he_normal', name='l3Conv1')(
        pool2)
    spDrop3 = SpatialDropout2D(.2)(l3Conv1)
    l3Conv2 = Conv2D(filters * 4, 3, activation='relu', padding='same',
kernel_initializer='he_normal', name='l3Conv2')(
        spDrop3)
    pool3 = MaxPooling2D(pool_size=(2, 2))(l3Conv2)

    l4Conv1 = Conv2D(filters * 8, 3, activation='relu', padding='same',
kernel_initializer='he_normal', name='l4Conv1')(
        pool3)
    spDrop4 = SpatialDropout2D(.2)(l4Conv1)
    l4Conv2 = Conv2D(filters * 8, 3, activation='relu', padding='same',
kernel_initializer='he_normal', name='l4Conv2')(
        spDrop4)
    pool4 = MaxPooling2D(pool_size=(2, 2))(l4Conv2)

    l5Conv1 = Conv2D(filters * 16, 3, activation='relu',
padding='same', kernel_initializer='he_normal',
                     name='l5Conv1')(pool4)
    spDrop5 = SpatialDropout2D(.2)(l5Conv1)
    l5Conv2 = Conv2D(filters * 16, 3, activation='relu',
```

```
padding='same', kernel_initializer='he_normal',
                      name='l5Conv2')(
        spDrop5)

    l6Up = Conv2DTranspose(filters * 8, (2, 2), strides=(2, 2),
padding='same')(l5Conv2)
    l6Conc = concatenate([l4Conv2, l6Up], axis=3)

    l6Conv1 = Conv2D(filters * 8, 3, activation='relu', padding='same',
kernel_initializer='he_normal')(l6Conc)
    spDrop6 = SpatialDropout2D(.2)(l6Conv1)
    l6Conv2 = Conv2D(filters * 8, 3, activation='relu', padding='same',
kernel_initializer='he_normal')(spDrop6)

    l7Up = Conv2DTranspose(filters * 4, (2, 2), strides=(2, 2),
padding='same')(l6Conv2)
    l7Conc = concatenate([l3Conv2, l7Up], axis=3)

    l7Conv1 = Conv2D(filters * 4, 3, activation='relu', padding='same',
kernel_initializer='he_normal')(l7Conc)
    spDrop7 = SpatialDropout2D(.2)(l7Conv1)
    l7Conv2 = Conv2D(filters * 4, 3, activation='relu', padding='same',
kernel_initializer='he_normal')(spDrop7)

    l8Up = Conv2DTranspose(128, (2, 2), strides=(2, 2),
padding='same')(l7Conv2)

    l8Conc = concatenate([l2Conv2, l8Up], axis=3)

    l8Conv1 = Conv2D(filters * 2, 3, activation='relu', padding='same',
kernel_initializer='he_normal')(l8Conc)
    spDrop8 = SpatialDropout2D(.2)(l8Conv1)
    l8Conv2 = Conv2D(filters * 2, 3, activation='relu', padding='same',
kernel_initializer='he_normal')(spDrop8)

    l9Up = Conv2DTranspose(64, (2, 2), strides=(2, 2),
padding='same')(l8Conv2)

    l9Conc = concatenate([l1Conv2, l9Up], axis=3)
    l9Conv1 = Conv2D(filters, 3, activation='relu', padding='same',
kernel_initializer='he_normal')(l9Conc)
    spDrop9 = SpatialDropout2D(.2)(l9Conv1)
    l9Conv2 = Conv2D(filters, 3, activation='relu', padding='same',
                      kernel_initializer='he_normal')(spDrop9)

    l9Conv3 = Conv2D(1, (1, 1), activation='sigmoid', padding='same',
kernel_initializer='he_normal')(l9Conv2)

    model = Model(inputs, l9Conv3)

    return model

# Fire Module
def Conv2D_Squeeze(xIn, filters):
    x = Conv2D(filters / 4, 1, activation='relu', padding='same',
kernel_initializer='he_normal')(xIn)
    x1 = Conv2D(filters/2, 1, activation='relu', padding='same',
```

```
kernel_initializer='he_normal')(x)
    x2 = Conv2D(filters/2, 3, activation='relu', padding='same',
kernel_initializer='he_normal')(x)
    xOut = concatenate((x1, x2), axis=3)
    return xOut

# Transposed Fire Module
def trans2D_Squeeze(xIn, filters):
    x = Conv2DTranspose(filters/4, 1,strides= (2,2),
padding='same')(xIn)
    x1 = Conv2DTranspose(filters/2, 1, padding='same')(x)
    x2 = Conv2DTranspose(filters/2, 2, padding='same')(x)
    xOut = concatenate((x1, x2), axis=3)
    return xOut


# U-Net Squeeze
def unetSqueeze(filters=64, input_size=(512, 512, 3)):
    inputs = Input(input_size)
    d1conv1 = Conv2D(filters, 3, activation='relu', padding='same',
kernel_initializer='he_normal')(inputs)
    d1drop = SpatialDropout2D(.2)(d1conv1)
    d1conv2 = Conv2D(filters, 3, activation='relu', padding='same',
kernel_initializer='he_normal')(d1drop)
    d1pool = MaxPooling2D(pool_size=(2, 2))(d1conv2)


    d2conv1 = Conv2D_Squeeze(d1pool, filters*2)
    d2drop = SpatialDropout2D(.2)(d2conv1)
    d2conv2 = Conv2D_Squeeze(d2drop, filters*2)
    d2pool = MaxPooling2D(pool_size=(2, 2))(d2conv2)

    d3conv1 = Conv2D_Squeeze(d2pool, filters * 4)
    d3drop = SpatialDropout2D(.2)(d3conv1)
    d3conv2 = Conv2D_Squeeze(d3drop, filters * 4)
    d3pool = MaxPooling2D(pool_size=(2, 2))(d3conv2)


    d4conv1 = Conv2D_Squeeze(d3pool, filters * 8)
    d4drop = SpatialDropout2D(.2)(d4conv1)
    d4conv2 = Conv2D_Squeeze(d4drop, filters * 8)
    d4pool = MaxPooling2D(pool_size=(2, 2))(d4conv2)

    d5conv1 = Conv2D_Squeeze(d4pool, filters * 16)
    d5drop = SpatialDropout2D(.2)(d5conv1)
    d5conv2 = Conv2D_Squeeze(d5drop, filters * 16)
    d5trans = trans2D_Squeeze(d5conv2, filters * 8)

    u1conc = concatenate([d4conv2, d5trans], axis=3)
    u1conv1 = Conv2D_Squeeze(u1conc, filters * 8)
    u1drop = SpatialDropout2D(.2)(u1conv1)
    u1conv2 = Conv2D_Squeeze(u1drop, filters * 8)
    u1trans = trans2D_Squeeze(u1conv2, filters * 4)

    u2conc = concatenate([d3conv2, u1trans], axis=3)
    u2conv1 = Conv2D_Squeeze(u2conc, filters * 4)
    u2drop = SpatialDropout2D(.2)(u2conv1)
```

```
    u2conv2 = Conv2D_Squeeze(u2drop, filters * 4)
    u2trans = trans2D_Squeeze(u2conv2, filters * 2)

    u3conc = concatenate([d2conv2, u2trans], axis=3)
    u3conv1 = Conv2D_Squeeze(u3conc, filters * 2)
    u3drop = SpatialDropout2D(.2)(u3conv1)
    u3conv2 = Conv2D_Squeeze(u3drop, filters * 2)
    u3trans = Conv2DTranspose(filters, 3, strides=(2, 2),
padding='same')(u3conv2)

    u4conc = concatenate([d1conv2, u3trans], axis=3)
    u4conv1 = Conv2D(filters, 3, activation='relu', padding='same',
kernel_initializer='he_normal')(u4conc)
    u4drop = SpatialDropout2D(.2)(u4conv1)
    u4conv2 = Conv2D(filters, 3, activation='relu', padding='same',
kernel_initializer='he_normal')(u4drop)
    u4conv3 = Conv2D(1, (1, 1), activation='sigmoid', padding='same',
kernel_initializer='he_normal')(u4conv2)

    model = Model(inputs, u4conv3)
    return model

# U-Net Large Module
def Conv2D_ModL(xIn, filters):
    x = Conv2D(filters / 4, 1, activation='relu', padding='same',
kernel_initializer='he_normal')(xIn)
    x1 = Conv2D(filters*3/4, 1, activation='relu', padding='same',
kernel_initializer='he_normal')(x)
    x2 = Conv2D(filters*1/4, 7, activation='relu', padding='same',
kernel_initializer='he_normal')(x)
    xOut = concatenate((x1, x2), axis=3)
    return xOut

# U-Net with large convolutions
def unetL(filters=64, input_size=(512, 512, 3)):
    inputs = Input(input_size)
    d1conv1 = Conv2D(filters, 3, activation='relu', padding='same',
kernel_initializer='he_normal')(inputs)
    d1drop = SpatialDropout2D(.2)(d1conv1)
    d1conv2 = Conv2D(filters, 3, activation='relu', padding='same',
kernel_initializer='he_normal')(d1drop)
    d1pool = MaxPooling2D(pool_size=(2, 2))(d1conv2)


    d2conv1 = Conv2D_ModL(d1pool, filters*2)
    d2drop = SpatialDropout2D(.2)(d2conv1)
    d2conv2 = Conv2D_ModL(d2drop, filters*2)
    d2pool = MaxPooling2D(pool_size=(2, 2))(d2conv2)

    d3conv1 = Conv2D_ModL(d2pool, filters * 4)
    d3drop = SpatialDropout2D(.2)(d3conv1)
    d3conv2 = Conv2D_ModL(d3drop, filters * 4)
    d3pool = MaxPooling2D(pool_size=(2, 2))(d3conv2)


    d4conv1 = Conv2D_ModL(d3pool, filters * 8)
    d4drop = SpatialDropout2D(.2)(d4conv1)
```

```
    d4conv2 = Conv2D_ModL(d4drop, filters * 8)
    d4pool = MaxPooling2D(pool_size=(2, 2))(d4conv2)

    d5conv1 = Conv2D_ModL(d4pool, filters * 16)
    d5drop = SpatialDropout2D(.2)(d5conv1)
    d5conv2 = Conv2D_ModL(d5drop, filters * 16)
    d5trans = trans2D_Squeeze(d5conv2, filters * 8)

    u1conc = concatenate([d4conv2, d5trans], axis=3)
    u1conv1 = Conv2D_ModL(u1conc, filters * 8)
    u1drop = SpatialDropout2D(.2)(u1conv1)
    u1conv2 = Conv2D_ModL(u1drop, filters * 8)
    u1trans = trans2D_Squeeze(u1conv2, filters * 4)

    u2conc = concatenate([d3conv2, u1trans], axis=3)
    u2conv1 = Conv2D_ModL(u2conc, filters * 4)
    u2drop = SpatialDropout2D(.2)(u2conv1)
    u2conv2 = Conv2D_ModL(u2drop, filters * 4)
    u2trans = trans2D_Squeeze(u2conv2, filters * 2)

    u3conc = concatenate([d2conv2, u2trans], axis=3)
    u3conv1 = Conv2D_ModL(u3conc, filters * 2)
    u3drop = SpatialDropout2D(.2)(u3conv1)
    u3conv2 = Conv2D_ModL(u3drop, filters * 2)
    u3trans = Conv2DTranspose(filters, 3, strides=(2, 2),
padding='same')(u3conv2)

    u4conc = concatenate([d1conv2, u3trans], axis=3)
    u4conv1 = Conv2D(filters, 3, activation='relu', padding='same',
kernel_initializer='he_normal')(u4conc)
    u4drop = SpatialDropout2D(.2)(u4conv1)
    u4conv2 = Conv2D(filters, 3, activation='relu', padding='same',
kernel_initializer='he_normal')(u4drop)
    u4conv3 = Conv2D(1, (1, 1), activation='sigmoid', padding='same',
kernel_initializer='he_normal')(u4conv2)

    model = Model(inputs, u4conv3)
    return model
```

Appendix D Downstream Tasks (Altitude Recovery etc.)

```
import cv2
import numpy
import numpy as np
import time
import math


# Simple binding to automatically transfer deg to rad. Simplifies
equations a lot in the code
def cos(x):
    return math.cos(math.radians(x))


# Simple binding to automatically transfer deg to rad. Simplifies
equations a lot in the code
def sin(x):
    return math.sin(math.radians(x))


# Returns corners from a binary mask
# INPUTS
#   mask NumPy array, binary mask in
#   timeIn int, time in seconds before loop breaks and returns nothing
#   img NumPy array, where to draw image
#   drawImg, boolean, whether to draw image
def getCorners(mask, timeIn, img=None, drawImg=False):
    contours, hierarchy = cv2.findContours(mask.astype(np.uint8),
cv2.RETR_TREE, cv2.CHAIN_APPROX_NONE)
    if len(contours) == 0:
        print("NO CONTOURS")
        return None

    if drawImg:
        cv2.drawContours(img, contours, -1, (0, 255, 0), 3)
        cv2.drawContours(img, contours, 2, (0, 255, 0), 10)

    # Sort all contours by area, highest area first
    cnt = sorted(contours, key=cv2.contourArea, reverse=True)

    # Runway should be largest by area
    cornerPolyLine = cnt[0]

    if drawImg:
        cv2.drawContours(img, cornerPolyLine, -1, (0, 255, 0), 3)

    # Perimeter of contour
    peri = cv2.arcLength(cornerPolyLine, True)
```

```
    start = time.time()
    timeTaken = 0
    eps = .002
    newList = list()

    lowestY = 512

    for corner in cornerPolyLine:
        # If corner is not within 10 pixels of edges
        if not (corner[0][0] < 10 or corner[0][0] > 512 - 10 or
corner[0][1] < 10 or corner[0][1] > 512 - 10
                or lowestY < corner[0][1]):
            newList.append(corner)
        # Else save the lowest row encountered. only works if camera
pitch <90 degrees
        elif (corner[0][0] < 10 or corner[0][0] > 512 - 10) and lowestY
> corner[0][1]:
            lowestY = corner[0][1]

    newList2 = list()
    # Removes all pixels below the lowest y value runway pixel that was
not near the edge
    for corner in newList:
        if not (lowestY < corner[0][1]):
            newList2.append(corner)

    newArr = numpy.array(newList2)

    # Slowly remove all extraneous edge pixels
    while timeTaken < timeIn:

        corners = cv2.approxPolyDP(newArr, eps * peri, True)

        # Reset perimeter to smaller contour's
        peri = cv2.arcLength(corners, True)
        timeTaken = time.time() - start
        eps += .001

        if timeTaken >= timeIn and not len(corners) < 8:
            print("TIME")
            return None

        elif corners.shape[0] == 4:
            return corners
        # If epsilon gets too large, the approximation will be
terrible, return none and print to console
        elif eps > .2:
            print("not clear")
            return np.array([None])


# Returns corners in the correct format for downstream tasks.
# INPUTS
#   cornersIn NumPy array, four corners in
#   edgeDim int, edge of image
#   inline boolean, if runway is inline with camera (width might be
```

```
larger than length in image)
#   drawImg, boolean, whether to draw image
def cleanCorners(cornersIn, edgeDim, inline):
    idxMin = -1
    distMin = -1

    if inline == True:

        cList = list()
        for corner in cornersIn:
            cList.append(corner[0])
        p1 = sorted(cList, key=lambda tup: tup[1])[0:2]
        p2 = sorted(cList, key=lambda tup: tup[1])[2:4]

        # If runway is completely inline with runway, points should be
horizontal
        if abs((p1[0][1] - p1[1][1]) / (p1[0][0] - p1[1][0])) < .1:
            return p1, p2, True

        else:
            return p1, p2, False


    else:

        # Gets closest corner to first corner
        for i in range(3):
            dist = math.pow(math.pow(cornersIn[0, 0][0] - cornersIn[i +
1, 0][0], 2) + math.pow(
                cornersIn[0, 0][1] - cornersIn[i + 1, 0][1], 2), .5)
            if distMin == -1 or distMin > dist:
                idxMin = i + 1
                distMin = dist
        secondPair = [1, 2, 3]
        secondPair.remove(idxMin)
        edge1 = False
        edge2 = False

        # If first corner and closest to first are near the edges
        if (cornersIn[0, 0][0] < 11 or cornersIn[0, 0][0] > edgeDim -
11) \
                or (cornersIn[0, 0][1] < 11 or cornersIn[0, 0][1] >
edgeDim - 11) \
                or cornersIn[idxMin, 0][0] < 11 or cornersIn[idxMin,
0][0] > edgeDim - 11 \
                or (cornersIn[idxMin, 0][1] < 11 or cornersIn[idxMin,
0][1] > edgeDim - 11):
            edge1 = True
        # If not first/not closest are near the edges
        if (cornersIn[secondPair[0], 0][0] < 11 or
cornersIn[secondPair[0], 0][0] > edgeDim - 11) \
                or (cornersIn[secondPair[0], 0][1] < 11 or
cornersIn[secondPair[0], 0][1] > edgeDim - 11) \
                or cornersIn[secondPair[1], 0][0] < 11 or
cornersIn[secondPair[1], 0][0] > edgeDim - 11 \
                or (cornersIn[secondPair[1], 0][1] < 11 or
cornersIn[secondPair[1], 0][1] > edgeDim - 11):
```

```
            edge2 = True

        # Determining which pair is further from camera and if it is
near the edge
        # Returns corners that are closest to camera, but not near the
edges
        if cornersIn[0, 0][1] > max(cornersIn[secondPair[0], 0][1],
cornersIn[secondPair[1], 0][1]) and not edge1:

            return cornersIn[0, 0], cornersIn[idxMin, 0]
        elif cornersIn[0, 0][1] > max(cornersIn[secondPair[0], 0][1],
                                      cornersIn[secondPair[1], 0][1])
and edge1 and edge2:

            return cornersIn[0, 0], cornersIn[idxMin, 0]

        elif edge2 and not edge1:

            return cornersIn[0, 0], cornersIn[idxMin, 0]
        else:

            return cornersIn[secondPair[0], 0],
cornersIn[secondPair[1], 0]


# From pixel U, P values to real plane, X, Y
# INPUTS:
#   altitude double, altitude of drone  (mm)
#   pitch double, pitch of drone (degrees)
#   Up double (or int), pixel U value that has been centered
#   Vp double (or int), pixel V value that has been centered
#   focus double, focal length of camera (mm)
#   sensorW double, sensor width in camera (mm)
#   sensorH double, sensor height in camera (mm)
#   pixW int, image width
#   pixH int, image height
def pixelToDistance(altitude, pitch, Up, Vp, focus, sensorW, sensorH,
pixW, pixH):

    # lens height above homogenous plane
    lensDist = ((altitude) / cos(90 - pitch))

    # Centering pixels and going from pixels to mm
    Ui = Up * (sensorW / pixW) - sensorW / 2
    Vi = (sensorH / 2) - Vp * (sensorH / pixH)

    # parameter t where ray is equal to real plane
    t = sin(pitch) * lensDist / (sin(pitch) * focus - cos(pitch) * Vi)

    # Homogenous coordinates at intercept
    Xh = Ui * t
    Yh = Vi * t

    # offset of altitude recording and lens homogenous center
    ydistOffset = lensDist * sin(90 - pitch)

    # Homogenous to real
```

```
    Yr = Yh / (cos(90 - pitch)) + ydistOffset
    Xr = Xh
    return Xr, Yr


# From Real X, Y to pixel centered U,V
# INPUTS:
#   altitude double, altitude of drone  (mm)
#   pitch double, pitch of drone (degrees)
#   x double (or int), real world x value from drone
#   y double (or int), real world y value from drone
#   focus double, focal length of camera (mm)
#   sensorW double, sensor width in camera (mm)
#   sensorH double, sensor height in camera (mm)
#   pixW int, image width
#   pixH int, image height
def realToPixel(altitude, pitch, x, y, focus, sensorW, sensorH, pixW,
pixH):
    # lens height above homogenous plane
    lensDist = ((altitude) / cos(90 - pitch))
    # offset of altitude recording and lens homogenous center
    ydistOffset = lensDist * sin(90 - pitch)

    Yr = y - ydistOffset
    Xr = x

    # Real to homogenous
    xh = Xr
    yh = cos(90 - pitch) * Yr
    zh = -sin(90 - pitch) * Yr

    # parameter t where ray is equal to image plane
    t = (lensDist - focus - zh) / (lensDist - zh)

    # Parameter to pixel values
    Ui = -xh * t + xh
    Vi = -yh * t + yh
    Zr = (lensDist - zh) * t + zh

    # Just a sanity check, never actually thrown in any tests
    if abs(Zr - (lensDist - focus)) > .00001:
        raise ValueError("Zr must be = lensDist - focus but difference
was %.5f" % (Zr - (lensDist - focus)))

    # Image plane values (mm) to pixel values and un centering
    Up = (Ui + sensorW / 2) * (pixW / sensorW)
    Vp = (1 - Vi / (sensorH / 2)) * pixH / 2
    return Up, Vp


# From two lines defined by points to altitude
# INPUTS:
#   pitch double, pitch of drone (degrees)
#   dimIn double, known width
#   l1 NumPy array, two point values defining line 1
#   l2 NumPy array, two point values defining line 2
#   focus double, focal length of camera (mm)
```

```
#   sensorW double, sensor width in camera (mm)
#   sensorH double, sensor height in camera (mm)
#   pixW int, image width
#   pixH int, image height
def altFromRealAndPixelSlope(pitch, dimIn, l1, l2, focus, sensorW,
sensorH, pixW, pixH):

    # Centering pixels and going from pixels to mm
    l1[:, 1] = (sensorH / 2) - l1[:, 1] * (sensorH / pixH)
    l2[:, 1] = (sensorH / 2) - l2[:, 1] * (sensorH / pixH)

    # Centering pixels and going from pixels to mm
    l1[:, 0] = l1[:, 0] * (sensorW / pixW) - sensorW / 2
    l2[:, 0] = l2[:, 0] * (sensorW / pixW) - sensorW / 2

    # Parameter t without h
    m11 = sin(pitch) / (sin(pitch) * focus - cos(pitch) * l1[0][1])
    m12 = sin(pitch) / (sin(pitch) * focus - cos(pitch) * l1[1][1])

    # Parameter t without h
    m21 = sin(pitch) / (sin(pitch) * focus - cos(pitch) * l2[0][1])
    m22 = sin(pitch) / (sin(pitch) * focus - cos(pitch) * l2[1][1])

    # Vertical Checking the slopes
    # If slopes would cause a div/0 error, then the lines are vertical
    # distance between lines is then used as the width
    if abs(l1[1][0] * m12 - l1[0][0] * m11) < .000001 and abs(l2[1][0]
* m22 - l2[0][0] * m21) < .001:
        diff = abs((l1[1][0] * m12 - l2[1][0] * m22))

        a1 = abs(dimIn / (diff))

        return (a1 * cos(90 - pitch)), (a1 * cos(90 - pitch)), (a1 *
cos(90 - pitch))

    # Slopes
    sl1 = (l1[1][1] * m12 - l1[0][1] * m11) / (cos(90 - pitch) *
(l1[1][0] * m12 - l1[0][0] * m11))
    sl2 = (l2[1][1] * m22 - l2[0][1] * m21) / (cos(90 - pitch) *
(l2[1][0] * m22 - l2[0][0] * m21))

    # Absolute value of the differences in b from line equation mx+b=y
    bAbs = abs(((l1[1][1] * m12 / cos(90 - pitch)) + sin(90 - pitch) -
sl1 * l1[1][0] * m12)
               - ((l2[1][1] * m22 / cos(90 - pitch)) + sin(90 - pitch)
- sl2 * l2[1][0] * m22))

    # if differences in parameters is 0, special
    # Rarely encountered and not encountered at all in the math
    # This happens when the neural network prediction is bad and pitch
reading is off
    # Error is a tiny bit higher than normal
    # Hit once out of most recent 100 image run
    if bAbs < .001:
        diff1 = abs((l1[1][0] * m12 - l2[1][0] * m22))
        diff2 = abs((l1[1][0] * m12 - l2[1][0] * m22))
        a1 = abs(dimIn / (diff1))
```

```
        a2 = abs(dimIn / (diff2))
        a3 = abs(dimIn / (diff2 + diff1) / 2)
        print("bprob")
        return (a1 * cos(90 - pitch)), (a2 * cos(90 - pitch)), (a3 *
cos(90 - pitch))

    # Three altitude values based on distance between parallel lines
    # Using first slope, second slope, and average
    a1 = dimIn * math.pow(1 + math.pow(sl1, 2), .5) / bAbs
    a2 = dimIn * math.pow(1 + math.pow(sl2, 2), .5) / bAbs
    a3 = dimIn * math.pow(1 + math.pow((sl1 + sl2) / 2, 2), .5) / bAbs

    return a1 * cos(90 - pitch), a2 * cos(90 - pitch), a3 * cos(90 -
pitch)

# Polyline four points to lines
# INPUTS:
#   cornersIn NumPy array, four corners in
#   method "close" or any string, if the returned lines should be the
points that are closest
#         always used as not close
def polyToPairs(cornersIn, method='close'):
    idxMin = -1
    distMin = -1

    # Get the corners that are closest to each other
    for i in range(3):
        dist = math.pow(math.pow(cornersIn[0, 0][0] - cornersIn[i + 1,
0][0], 2) + math.pow(
            cornersIn[0, 0][1] - cornersIn[i + 1, 0][1], 2), .5)
        if distMin == -1 or distMin > dist:
            idxMin = i + 1
            distMin = dist
    secondPair = [1, 2, 3]
    secondPair.remove(idxMin)

    # Set up lines based on these points
    l1 = [cornersIn[0, 0], cornersIn[idxMin, 0]]
    l2 = [cornersIn[secondPair[0], 0], cornersIn[secondPair[1], 0]]

    # return the pairs of points that are closest
    if method == "close":
        return np.array(l1), np.array(l2)

    else:
        l3 = np.zeros([2, 2])
        l4 = np.zeros([2, 2])

        # Pairs are instead based on left/rightness
        # This method falls apart quite easily if the runway is at a
weird angle to the camera
        # Next step would be to sort corners using the method in utils
and base the pairs from that sorting
        # The two videos used for testing do not fall into this
issue... all videos where it occurs was used in training
        # Besides more training data, this will be the first thing to
be worked on
```

```
        l3[0], l4[0] = sorted(l1, key=lambda tup: tup[0])
        l3[1], l4[1] = sorted(l2, key=lambda tup: tup[0])
        return np.array(l3), np.array(l4)

# From two points to altitude
# INPUTS:
#   dimIn double, known width
#   pitch double, pitch of drone (degrees)
#   p1 NumPy array, one corner
#   p2 NumPy array, one corner
#   focus double, focal length of camera (mm)
#   sensorW double, sensor width in camera (mm)
#   sensorH double, sensor height in camera (mm)
#   pixW int, image width
#   pixH int, image height
def altFromTwoCorners(dimIn, pitch, p1, p0, focus, sensorW, sensorH,
pixW, pixH):

    # Pixel to image plane (mm)
    U1i = p1[0] * (sensorW / pixW) - sensorW / 2
    V1i = (sensorH / 2) - p1[1] * (sensorH / pixH)
    U0i = p0[0] * (sensorW / pixW) - sensorW / 2
    V0i = (sensorH / 2) - p0[1] * (sensorH / pixH)

    # Parameter t without h
    M1 = sin(pitch) / (sin(pitch) * focus - cos(pitch) * V1i)
    M0 = sin(pitch) / (sin(pitch) * focus - cos(pitch) * V0i)

    # altitude equation, please see thesis for derivation
    h = dimIn / math.pow(
        math.pow(U1i * M1 - U0i * M0, 2) + math.pow(V1i * M1 * (1 /
cos(90 - pitch)) - V0i * M0 * (1 / cos(90 - pitch)),
                                                    2), .5)

    return h * cos(90 - pitch)
```