Machine Learning with User-Friendly Data Generation for Computer Vision-based Navigation

A Major Qualifying Project

submitted to the faculty of

WORCESTER POLYTECHNIC INSTITUTE

in partial fulfillment of the requirements for the

Degree in Bachelor of Science in Computer Science, Data Science, and Mathematical Sciences

by

Ryan Firenze

Margaret Munroe

Sebastian Pineda

Amos Roche

Arman Saduakas

Report Submitted to:

Prof. Oren Mangoubi

Prof. Gregory Noetscher

WPI

I. Abstract

The military utilizes parafoil parachutes to deliver munitions and other supplies to soldiers in the field. These parachutes must land in precise locations, and in the past the Global Positioning System (GPS) has been used to ensure that these deliveries reach their targets. However, some issues arise when using GPS. It can be tracked by others, jammed, or spoofed. As these deliveries contain important supplies, it is critical that methods are developed to prevent these instances of interference.

Last year a simulator was developed that allows users to determine the location of these parachutes as they fall based on video footage from a camera mounted on a parachute [1]. We are utilizing this simulator to collect simulated video footage to feature-match images from the simulation to determine the parachute's change in position during the period between screen captures. We used image recognition methods to recognize items and regions within the footage, and this will allow users to track where the parachute is in relation to the map. This will be a useful feature of the simulator, as it will allow users to direct the parafoils to land in specific locations (for example, within a boundary fence or near a body of water).

We developed a more intuitive user interface (UI) for the simulator. We added options for users to control various aspects of the simulation without the need for intimate knowledge of the program. Among these we will include the start location, flight duration, and flight path of the parachute, as well as weather conditions and time of day. The UI will also allow users to pause the simulation at any time, and output images at a predetermined time interval based on the number of images desired and the

duration of the simulation. Lastly, we added a second simulator that allows users to simulate a horizontal flight, rather than a vertical drop, including features that allow the simulation to be paused and the camera angle to be changed during the flight.

Lastly, we created two machine learning models. The first was a transfer learning model [2] which was able to take high altitude images as an input and segment parts of the image based on geographic regions, such as sea, land, and manmade roads. The second was a Convolutional Neural Network [3] that could take in the labeled images from the transfer learning model in pairs and predict how much position (in altitude, longitude, and latitude) has changed between the two photos.

II. Executive Summary

The U.S. Army Combat Capabilities Development Command (DEVCOM) is the United States' Army technology developer. The DEVCOM Soldier Center (DEVCOM-SC) is in charge with supplying the Army with life support and field feeding systems, clothing, precision airdrop systems, and ballistic, chemical, and laser protection systems. DEVCOM-SC commonly uses parafoils, a type of parachute, to deliver these packages to their destination [1]. These parafoils can use a Global Navigation System (GPS) to maneuver and land at their desired destination. However, GPS can fail during a parafoil drop due to any number of reasons, such as atmospheric changes, weather conditions, or even active interference such as GPS jamming and spoofing [4]. As such, our team was tasked with developing a program that could be utilized to properly maneuver a parafoil with a package payload without the use of a GPS.

Figure 1: DEVCOM uses parafoils to deliver supplies in areas that are difficult to reach otherwise. [5]

We had two main methods to approaching this problem. First, we wanted to develop a machine learning model that could recognize different geographic areas in an image. In practice, this could be used by a camera mounted on a parafoil taking in video feed of its descent. Each frame of this video could then be used by our model to recognize areas in the image which could in turn be used to determine which area would be best to land. In tandem to developing a machine learning model, we also wanted to continue development of the flight simulator worked on by last year's MQP team. The main goal for this simulation was to wrap it as a complete application that is downloadable and easily usable regardless of technical training. This simulation in its complete form utilizes the Cesium plug in for the Unreal engine which allows us to fully simulate geographic regions with real world coordinates. The program simulates a parachute dropping from a given start location for a given amount of time. We created a built-in screenshot feature that can export many images from this simulation. Herein lies the main purpose of this objective which was producing numerous quality images that we can use to train our machine learning model without the need to physically travel to those areas. This would allow DEVCOM-SC the ability to improve their parafoil dropping performance without the need to gather data from possibly unreachable areas.

We developed a neural network that can recognize regions within footage obtained from the simulator. It uses a method of image recognition known as semantic segmentation, which allows it to label every pixel in an image and assign a pre-determined class to each. We found that a convolutional neural network was the best suited to apply this to method to our data. Transfer learning was then used to improve

the model, as we provided labeled data that is like the input that it will be receiving. After testing this against several other methods that we tried, we found that our model that uses the Adam optimizer was the best, as it only had a mean squared error of 0.92, and a variance of 3.15. Since our program can recognize different regions of the screenshots from the simulator, we can determine the movement that took place between screenshots.

We also developed a user interface for the simulator. Previously, using the simulator required users to change the code to change specifications like location. We have provided users with a menu that allows them to choose a starting location, weather conditions, time of day, camera angle, number of screenshots, flight path, and duration of the flight. These options are provided in a nested menu of settings which can be found along with the play and quit buttons on the front page of the application. We have packaged the simulator and our improvements so it can be sent to other computers and used by whoever we send it to. A screenshot feature was added since the simulator is mainly used for data collection. Users decide how long they want the simulation to last and how many screenshots they want, and the program will take screenshots at the calculated interval. The simulation can also be paused, and when it is paused the rest of the screenshots are delayed so they are still at the same points they would be in the flight. Lastly, we developed a horizontal simulator that provides a different perspective than our main simulator, as the flight paths provided for the main simulator are those of parafoils released from planes. The horizontal simulator has the same features as the main simulator, except the flight path options as it always shows a direct flight.

III. Acknowledgments

We would like to thank our sponsor, advisor, the Mathematics, Computer Science, and Data Science departments, and last year's MQP group for their endless support. Firstly, we would like to thank Joseph Scheufele and Juliette Spitaels, two members of this MQP last year, for their support and guidance in starting our continuation of the project. Next, we would like to thank our sponsor Dr. Greg Noetscher and The United States Army Combat Capabilities Development Command (DEVCOM) for their expertise and aid in providing us with data and resources for the project. Without them, starting and completing this project may not have even been possible. Lastly, we would like to express our gratitude towards our project advisor, Professor Oren Mangoubi, for his help in starting the project and frequent support on how to continue to achieve our vision for this project.

IV. Table of Contents

V. List of Figures

1. Introduction

The United States military has a long history of using parafoils to deliver supplies to active-duty soldiers virtually anywhere in the world. The U.S. Army Combat Capabilities Development Command (DEVCOM) has been working to improve these kinds of deliveries, and one of the areas that is important to their success is the accuracy of these airdrops, as they contain supplies that are vital to military operations [1].

To ensure some level of accuracy, they have been using Global Positioning System (GPS). GPS was originally developed for military purposes but today is used for both civilian and its original purposes. However, there are problems with GPS as it can be interfered with in the forms of jamming and spoofing [4]. Severe weather can also impede connections of the GPS. These drawbacks have led DEVCOM to search for a different method of determining the location of their parafoils.

Last year DEVCOM sponsored an MQP that developed a simulator that allows them to test how their parafoils will fall in various locations. This allows them to test how an airdrop will be performed without wasting resources.

Our tasks included developing a user interface to make this simulator easier to use for DEVCOM and implementing a neural network that can recognize different regions of the footage from the simulator, as well as changes in position based on pairs of images from the footage. A better user interface will allow users to use it without having to change the code. Instead, they can just enter their desired values into textboxes and dropdown menus. Creating a machine learning model with the ability to

recognize various regions will be helpful for navigating these parachutes as we will then be able to use the screenshots from the simulator to determine how the parachute has moved.

1.1. Deliverables

We were tasked with developing an intuitive user interface for a payload drop simulator that was developed by last year's team since their simulator required either extensive knowledge of its inner workings or a comprehensive guide on how to build, download, and utilize it, we wanted to wrap the simulator into one easily usable program. Our interface allows users to decide on weather conditions, time of day, location, flight duration, and flight path.

Another goal of ours was to develop a program that can track and detect objects in footage from the payload's parachute. Our program tracks a chosen feature as the parachute falls and accounts for the rotation and movement of the parachute as it feature-matches between the original image and footage from the mounted camera. We first developed a program using a pretrained machine learning model that can recognize geographic locations such as a lake, pond, or other body of water. From there, we further trained and fine-tuned the model to recognize other chosen items, such as city roads, buildings, and other attributes of urban areas. The program recognizes and determines the location of an object within the frame and uses this information to determine the physical location based off the starting point for the parachute and flight path.

Finally, our team was tasked with developing a horizontal flight simulator. This allows users to simulate a direct flight from a chosen location to another. Whereas the payload drop simulation uses flight paths taken from actual parachute drops, which can move sporadically, the horizontal simulation gives a much smoother path, and this provides us with more useful footage. Also, since there does not need to be any altitude change when using this simulator, it is easier to implement our object detection program as we can see regions and objects on the ground and their size is not changing, so the scale will remain constant.

2. Background

After last year's team developed a simulator, we found that we could vastly improve the user interface by adding several useful functions. We also endeavored to add an ability to recognize certain objects in the simulator. In this section we discuss the history and previous work that guided our research, and the information that we found regarding image recognition methods.

2.1. DEVCOM-SC and Previous Work

The U.S. Army Combat Capabilities Development Command (DEVCOM) is the Army's technology developer. There are eight major competency areas within DEVCOM, and we are working with DEVCOM Soldier Center (DEVCOM-SC), the branch that provides the Army with life support and field feeding systems, clothing, precision airdrop systems, and ballistic, chemical, and laser protection systems. Many of their advancements in precision airdrop systems pertain to parafoils, a type of parachute that they use for supply delivery. The earliest parachutes date back to the 12th century, but the parafoil design as we know it was patented in the 1960s [6]. While these parachutes have been used with carts to carry a pilot, our project focuses on their use in delivering supplies to soldiers in the field. These parafoils fly unmanned, requiring only an input of a flight path.

Figure 2: The Joint Precision Airdrop System (JPADS) includes a parachute decelerator, an autonomous guidance unit and a load container. [5]

These parafoils can be guided using a Navigation Guidance and Control System [7]. This development has allowed the military to deliver their supplies more precisely, which is critical in many situations. The loads that are dropped contain an Airborne Guidance Unit (AGU) that stores software that aids in its direction and steering. Information provided by this software includes system positioning, velocity, and heading. Using these inputs, the system guides the load to the target location through a series of extensions and/or retractions of the control lines connected to the parachute [8].

In the past, GPS has been used to determine the location of the parafoil and guide it to the desired landing location. However, GPS can lose its signal because of atmospheric changes, weather changes, and solar activity (cite GPS signal loss). There

are also added risks when it is used to deliver supplies to the military, including jamming and spoofing. Jamming is a type of interference that blocks signals between satellites and receivers. Certain jamming devices are used by foreign militaries as well as inexpensive personal jammers. However, personal jammers are illegal in many countries. Spoofing is a different interference technique, in which a false signal is used to confuse a satellite receiver. When a spoofing signal is sent, the receiver will think it is somewhere other than its actual location (cite jamming and spoofing of GNSS). Both methods of interference are unacceptable, so DEVCOM has been working to find a method of locating and directing their parafoils that does not require GPS.

There have been two projects completed by WPI students in conjunction with DEVCOM with regards to alternate parachute navigation. DEVCOM aims to develop alternatives to GPS-based navigation, and these projects have helped them make progress in this area. The first team attempted to use footage from actual parafoil drops to develop a method for determining location [4]. After unsupervised learning proved unsuccessful, they used three feature matching algorithms in conjunction to determine the location of parachutes. However, they faced a lack of data that hindered their progress, since the military cannot afford to deploy enough parachutes to gather substantial data. To solve this problem, the project that was completed last year endeavored to create a simulator that would allow users to gather data without using actual planes and parachutes [1]. The synthetic data provided by the simulator includes labeled images that are then used as input for their deep learning systems.

2.2. Image Recognition Methods

We conducted extensive research on various image preprocessing methods such as coordinate location, and object detection methods including single shot detection, contour detection, edge detection, and semantic segmentation. We also investigated object tracking methods that function as object detection for videos rather than images.

Digital images are represented in a matrix format. Image processing is using mathematical algorithms and techniques to transform images to extract/retrieve valuable information from them. Common Image processing techniques include image enhancement, image segmentation, object detection, image classification/regression, and image compression. A common goal of image processing is to retrieve relevant information from the images. This involves masking out unwanted features in the image and analyzing the image's texture/color.

2.2.1 Types of Images

Binary Images have two unique values for pixel intensity values. The pixel value zero represents the color black and the value 255 represents the color white. Binary images are often used to mask unwanted features in the image. In Figure 4, the image histogram is shown for the image shown in Figure 3. A binary image histogram shows the distribution of the two-pixel values. In this image, it is shown by the histogram that there are more white pixels than black pixels in the image. Binary images are useful because objects are subtracted from the background easily and can allow for easy localization on the object. This allows for a lower computational

intensity for machine learning models because the binary image matrix only has two possible values. However, binary images can be problematic if image color is important for analysis.

Figure 3: Example of a Binary Image [9]

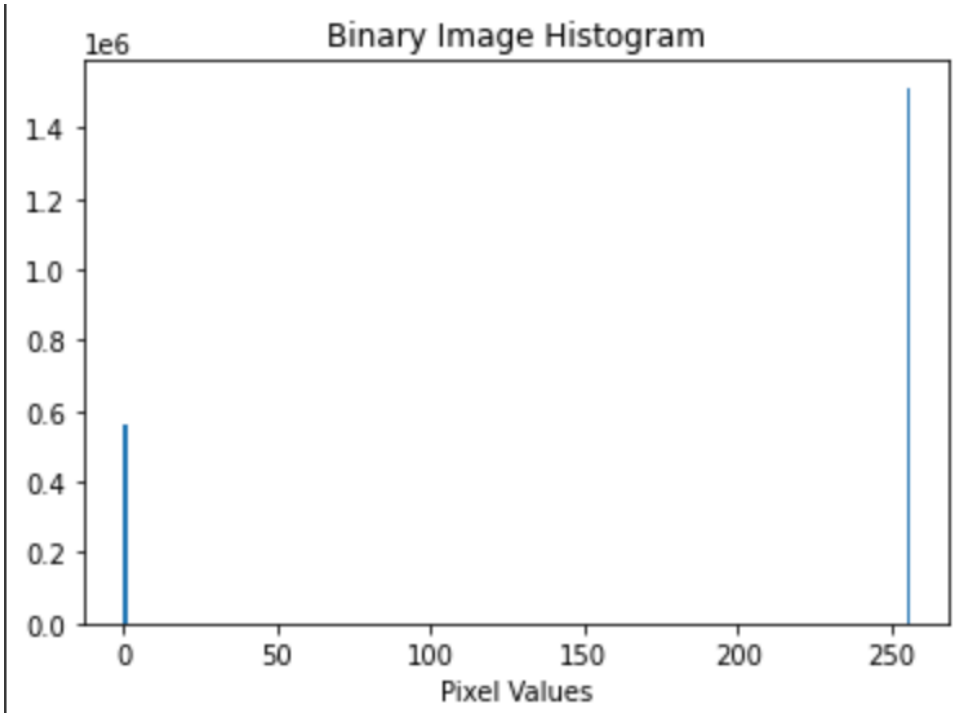


Figure 4: Image Histogram of Binary Image Pixel Values

Gray scale images consist of 256 unique colors where pixel values of zero represent black and values of 255 represent the color white. The pixel values between 1 and 254 represent different shades of gray. Grayscale images require three times less data than color images because RGB images require three dimensions, one for the red spectrum, one for the green spectrum, and one for the blue spectrum. Similarly, to Binary images, grayscale images are important because it can simplify the computational intensity for machine learning algorithms because of the lower amount of data needed to be processed. In Figure 6, the grayscale histogram shows most of the pixels are closer to the black shade.

Figure 5: Grayscale Image Representation [9]

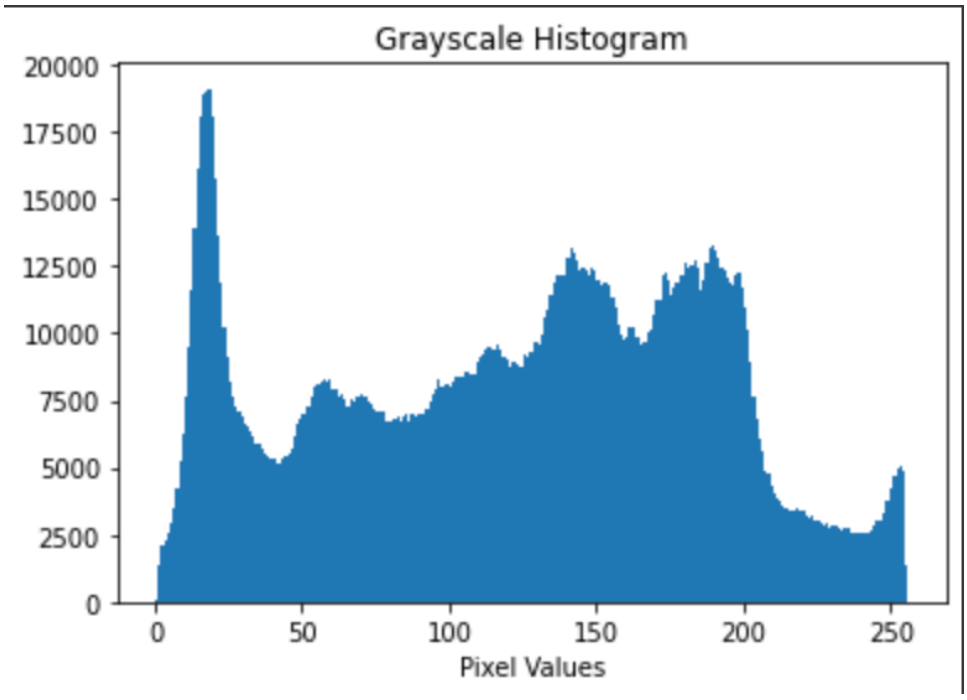


Figure 6: Grayscale Image Histogram

2.2.2 Image preprocessing

Image data preprocessing is necessary to convert the image data into suitable form for machine learning computations. Common data preprocessing steps include:

1. Blurring [10]
2. Normalization [11]
3. Data Augmentation [12]
4. Image Standardization [13]

Blurring is the process of reducing and distorting the detail of an image [10]. Blurring causes the noise to reduce in an image. Instead of focusing on small artifacts/objects, image blurring attempts to understand the whole structure of the image. Blurring is common in feature extraction to increase the visibility of certain textures and features in the image. It is also a common data augmentation technique to generate many blurred versions of an image. This can cause the machine learning model to be robust because the diversity of the training set increases from data augmentation. A common technique for image blurring is Gaussian filter blurring [10]. The filter looks to remove high-frequency features in the image while keeping low-level structural information. The filter is derived from the Gaussian function, the probabilistic distribution where the data is symmetric about the mean and can be visualized in Figure 7. The 2D Gaussian function is shown in Figure 8. The X and Y variables in the function represent the distances from center of the kernel in the horizontal/vertical directions. The variables value can be arbitrarily chosen. A common Gaussian kernel

shown in Figure 9a is convolved with the original image by sliding the kernel over the image and calculating the weighted sum of the pixel values in the 3x3 kernel neighborhood. If the kernel value is large, the image will appear to be blurrier which can be visualized in Figure 9b.

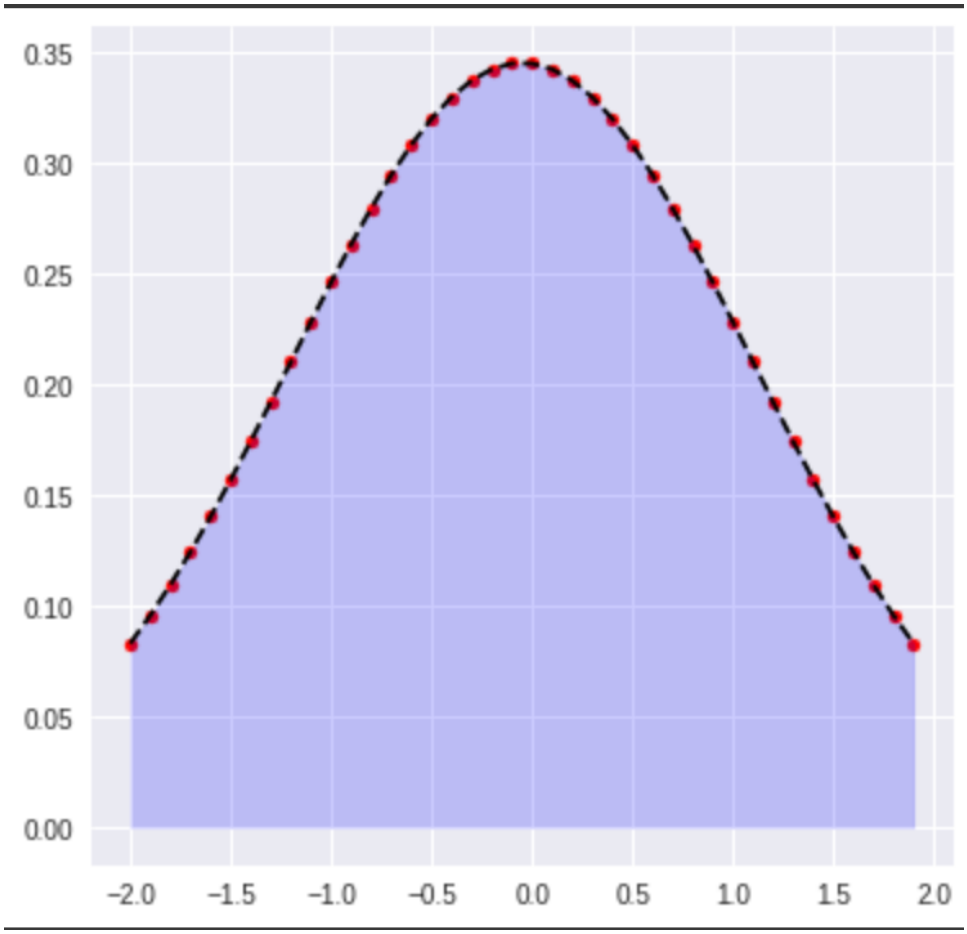


Figure 7: Gaussian "Bell Curve"

$$G(x,y) = \left(\frac{1}{\sigma\sqrt{2\pi}} e^{\left(-\frac{(x^2+y^2)}{(2\sigma^2)}\right)}\right)$$

Figure 8: Gaussian 2D Function formula

(a)

(b)

Figure 9: Image a represents a kernel value of 3 and image b represents a kernel value of 9.

Image normalization is a common preprocessing technique to transform an image into pixel values between zero and one [11]. This can result in better performance for machine learning algorithms because the range of pixel values is typically much lower in normalized form. Normalization also reduces effects of color/contrast variation in the image. Typically, the speed of training is quicker when trained with normalized pixel values. The first step in image normalization is to compute the mean and standard deviation of the image using the formulas outlined in Figure 10. Each of the *i* pixels can be normalized by using the mean and standard deviation values in the formula outlined in Figure 11.

$$\frac{\left(\mu - \left(\frac{1}{N}\right) \cdot \sum_{i=1}^{N}\left(p_i\right)\right)}{\left(\sigma = \left(\frac{1}{N}\right) \cdot \sum_{i=1}^{N}\left(p_i - \mu\right)\right)}$$

Figure 10: Formula computing the mean and standard deviation of an image pixel matrix where I represents the intensity of the ith pixel and N represents the number of pixels in the image [14].

$$p - hat = \frac{p_i - \mu}{\sigma}$$

Figure 11: Formula for normalizing the ith pixel [14].

Data augmentation is a technique of artificially increasing the training dataset by using techniques such as randomly rotating the images, changing the image's brightness, and stretching an image [12]. Augmentation is useful because acquiring large amounts of image label data is expensive and time consuming, so data

augmentation can generalize on a larger range of images easier. It also reduces overfitting problems because the training set is more diverse which can increase the model's ability to generalize for more situations.

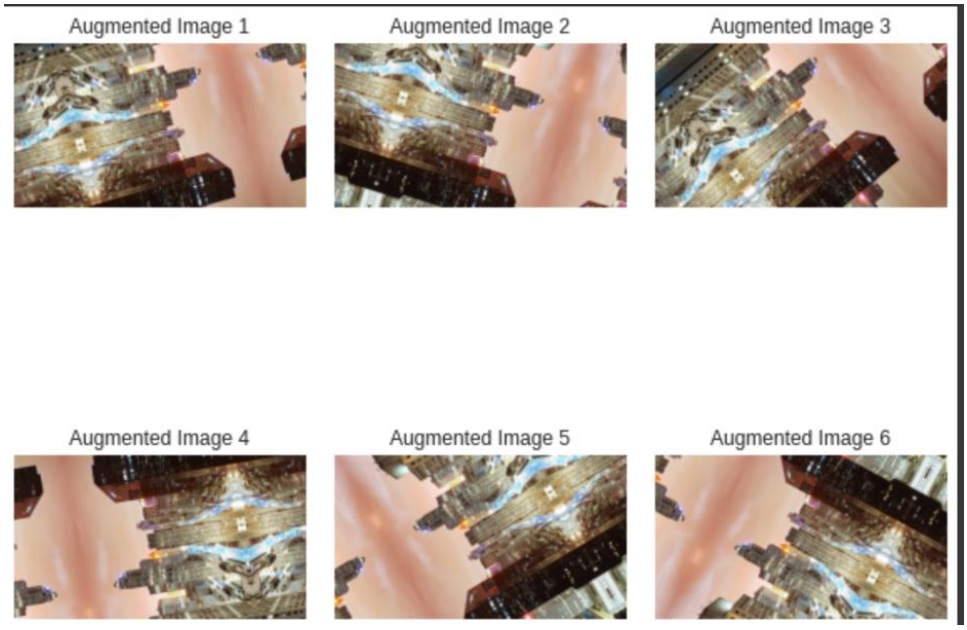
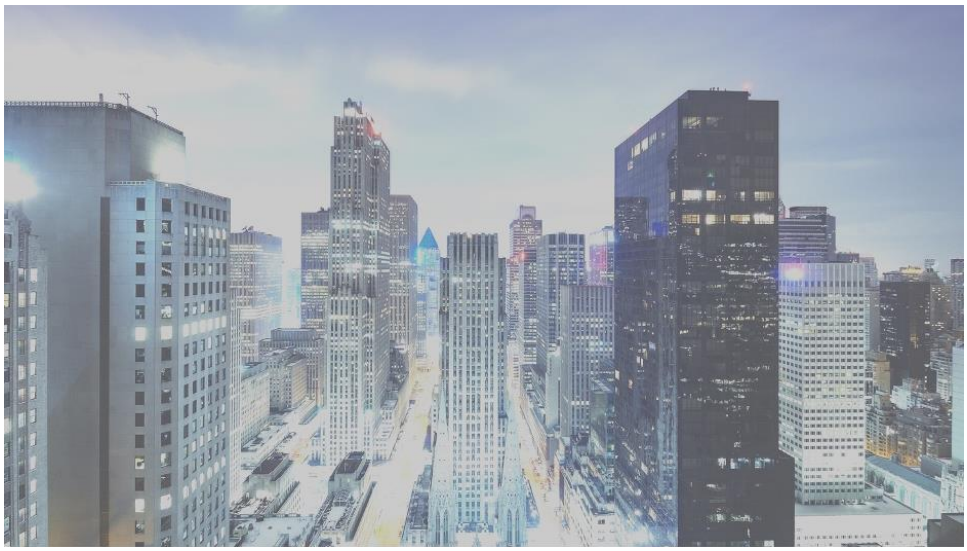


Figure 12: Random Rotation Data Augmentation.



a)

b)

c)

Figure 13: Images a), b), and c) all are randomly generated using the PyTorch Color Jitter Color Augmentation function. [9]

Image standardization is useful because it can reduce the variation of lighting/contrast between different images [13]. It can also result in faster machine learning model convergence and training because the scale and variance of the input data will decrease which allows for better stability/optimization of the machine learning

model. This technique also assures various features of the input data will be treated equally because the input features will be converted into the same scale with a mean of zero and standard deviation of one. An image can be standardized by subtracting the mean from an image pixel and dividing it by the standard deviation of the pixel values.

Image preprocessing methods that we researched were all parts of the OpenCV package on Python [15]. OpenCV includes the ability to read and show images and videos and several other functionalities. These include resizing and interpolation methods. Resizing an image is more complicated than just zooming in on it. If we have an image and we want to increase its size while maintaining the same resolution, we need to add pixels. An added pixel's color is chosen based on its surrounding pixels. Results of this method deteriorate the more the image is stretched. It is impossible to add details to the image if they are not present to begin with. Part of how this process is completed is with interpolation. If we have two data points, we can determine a line of best fit to find an appropriate value for any data point between them. When we have more than two data points, we can find a curve that fits the existing data and use this to find an unknown value. There are different classes of interpolation, including bilinear, linear, bicubic, and nearest neighbor methods [16]. Bilinear interpolation adds pixels by averaging the surrounding values, and similarly, the nearest neighbor method colors the new pixel by copying the one adjacent to it. Linear interpolation decides the value of the added pixel by choosing a color directly between the two existing pixels. Bicubic interpolation provides a smoother image, as it looks at the sixteen surrounding pixels instead of the four directly adjacent pixels. All these methods create data points, so none

of them are a true representation of the original image, but with the smoother image from bicubic interpolation, there are fewer pixels that appear out of place.

Contour detection is a method of detecting the borders of an object, allowing users to locate the object in an image [17]. It's a common technique that is used as a first step for both foreground extraction and simple image segmentation. Contours are boundaries between pixels of different color and intensity, so if we can detect them, we can easily locate objects within an image. To use this technique, we must first gray scale the image. The next step is to apply binary thresholding, which converts the image to black and white (only the two colors, as opposed to black and white photographs). Doing this in OpenCV will highlight objects of interest so the contour detection algorithm can easily find the borders. Once the contours are identified, they are overlayed on the original image.

Edge detection is a similar method, however, while contour detection finds boundaries which are closed curves, edge detection finds pixels with a significant difference from their neighbors, but these edges are not connected to form a contour. Edge detection is a technique in computer vision and image processing which involves identifying the edges in an image. John Canny, the creator of Canny Edge Detection states, "The edge detection process serves to simplify the analysis of images by drastically reducing the amount of data to be processed, while at the same time preserving useful structural information about object boundaries" [18]. Edge detection is often used as a pre-processing step for computer vision tasks because of image compression. Edge detection reduces the amount of data because if the edges are

identified, it will be possible to represent the image array with fewer pixels/dimensions without losing much information [19]. This ultimately leads to reducing the computational intensity of computer vision algorithms. The three most popular edge detection algorithms include the Canny Edge Detector, the Sobel operator, and the Laplacian of Gaussian (LoG) operator.

Sobel edge detection is a gradient based method which was developed in 1968 by Irwin Sobel and Gary Feldman [20]. The Sobel filter is used to calculate the gradient of the image intensity at each pixel in the image. The reason for this is to determine the direction/magnitude of the largest increase/decrease of the pixel intensity. When the Sobel filter is used for pixels with a constant intensity, those resulting pixels will be a zero vector [21]. But when the filter is applied to an edge, the resulting vector represents the points across the edge from the darker to brighter values. The Sobel filter consists of two 3x3 kernels which are represented in Figure 16, each of which represents the horizontal and vertical direction [21]. The X direction Sobel filter is multiplied by the original image source to find the increasing change in the right direction. Similarly, the Y Direction Sobel filter is multiplied by the original image source to find the increasing change in the downward direction. In Figure 15, it shows how a sample point b22 from the G_x (X directional gradient) is calculated by multiplying the input image by the kernel matrix. The same process is used to calculate the point from G_y (Y directional gradient). To combine these directional gradients for point b22 into one point we can use the magnitude formula in Figure 14. So, to find the magnitude for point b22, the formula from Figure 17 can be applied. This framework

can be followed for all the pixel points in the X-Y directions and the magnitude can be taken to get a final output edge detection image. In Figure 18, the Sobel X and Y direction output can be compared with the original image.

$$G = \sqrt{Gx^2 + Gy^2}$$

Figure 14: Calculates the magnitude of two directional pixel points.

$$\sqrt{b22x^2 + b22y^2}$$

Figure 15: Formula to calculate magnitude for point b22.

1	**0**	**-1**
2	0	-2
1	0	-1

a) X Direction Kernel

-1	**-2**	**-1**
0	0	0
1	2	1

b) Y Direction Kernel

Figure 16: a) represents the x direction kernel to calculate the X Directional gradient [20]. b) represents the Y direction kernel to calculate the Y Directional gradient.

Input Image

a11	**a12**	**a13**
a21	a22	a23
a31	a32	a33

Kernel Matrix

1	**0**	**-1**
2	0	-2
1	0	-1

Output Image

b11	**b12**	**b13**
b21	b22	b23
b31	b32	b33

b22 = a13-a11+2a23-2a21+a33-a31

Figure 17: Represents how the b22 pixel point is calculated by multiplying the input image by the kernel matrix for the X Direction [20].

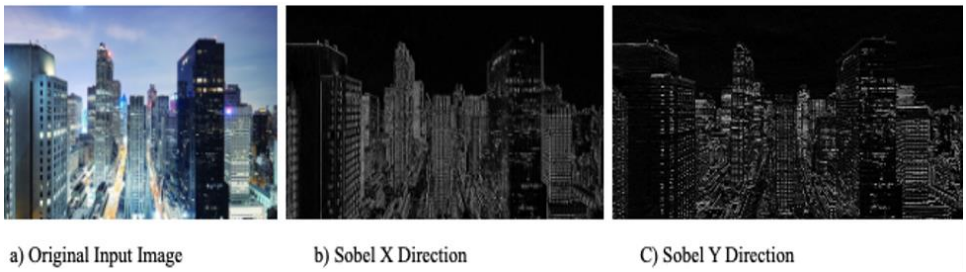


Figure 18: Shows the output of Sobel Detection vs the Original Image

Canny edge detection was developed in 1986 by John Canny [18]. The Canny edge detection algorithm utilizes a multi-stage approach to detect various edges in an image. The steps to perform Canny Edge Detection consist of:

1. Noise Reduction
2. Calculating the Gradient
3. Non-Maximal Suppression

4. Double Threshold
5. Edge Tracking

Noise Reduction is vital to calculate the gradients because extra noise can cause variance in the edge detection results. A common technique to perform noise reduction is the gaussian blur algorithm [10].

Next, the gradient is calculated to find the directional pixel intensity changes. The gradient calculated is the same as the Sobel Edge Detection gradient calculation.

After calculating the gradient, Non-Maximum Suppression is performed. The purpose of this algorithm is to make the edges thinner because thick edges can overlap with other structural information [22]. For example, if pixel A's image intensity is larger than two other pixels in the same gradient direction, the pixel will stay the same. Otherwise, if the pixel value is smaller, the pixel will be set to zero and it will be converted to a black pixel [22]. The goal of non-maximal suppression is to find the pixels with the maximum values in the edge directions. The other pixels will be converted into a black pixel value.

Double Thresholding looks to further decrease noise in the image and smooth out the edges [22]. Double Thresholding uses a high and low threshold. For example, if the high threshold is 0.7, normalized pixel values larger than 0.7 will have a strong edge. If the low threshold value is below 0.4, all normalized pixel values below 0.4 won't be an edge anymore. Edges between 0.3 and 0.7 will be classified as a weak pixel.

Finally, the last step is edge tracking which is used to determine which weak edges are real edges [22]. Weak edges which are connected to strong edges will be converted to a strong edge. Weak edges that aren't connected to a strong edge are removed. The result of the canny edge detection can be illustrated in Figures 19 and 20.

Figure 19: Gaussian Blur applied on the Input Image [9]

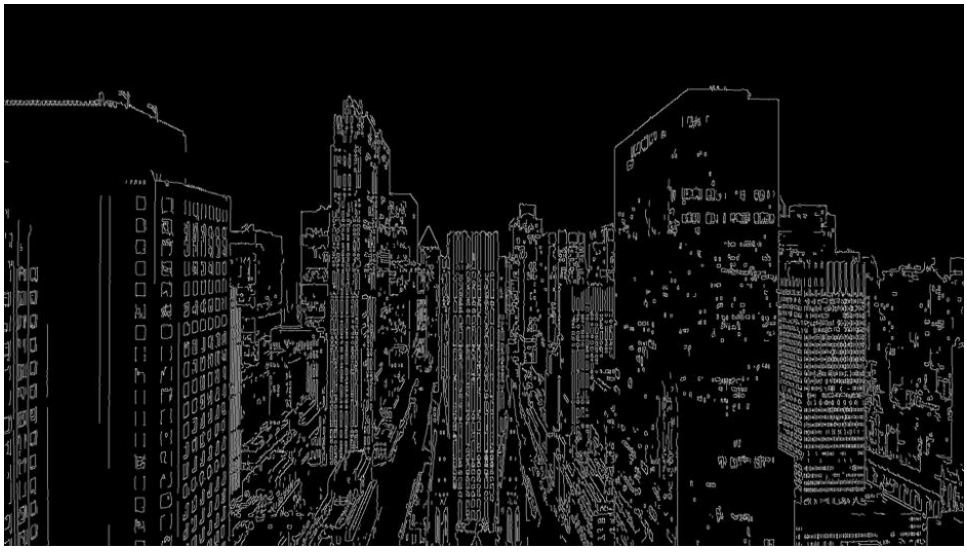


*Figure 20: Canny Edge De*tection Applied on the Blurred Image

Laplacian edge detection was introduced by Lawrence Roberts in 1963. The first step is to apply a gaussian filter to remove the noise and small details in the image [23]. The Laplacian of an image is computed using the equation outlined in Figure 21. This equation has been used to form a 3x3 discrete matrix approximation shown in Figure 22a. This 3x3 matrix is applied with the input image to produce an output which represents the second derivative of the input image. The output image will have positive pixel values where the intensity of the image is increasing and negative pixel values where the intensity of the image is decreasing. Lastly, a thresholding operator is applied to separate non-edges and edges [23]. A common technique for this is Otsu's method [24]. The idea behind this technique is to find the weighted average of the intensity values between the edge and non-edge classes. The between-class variance is then calculated using the formula outlined in Figure 22b. W_1 and W_2 represent the weights of the two classes. Mu_1 and Mu_2 represent the mean of the pixel intensity values of the two classes [23]. The ideal threshold is the value that maximizes the between-class variance between the non-edge and edge classes. The result of the Otsu's method is an image where the edge pixels are white and the non-edge pixels are black which can be seen in Figure 23.

$$L(x,y) = \frac{\partial^2 I(x,y)}{\partial x^2} + \frac{\partial^2 I(x,y)}{\partial y 2}$$

Figure 21: Second Derivative Laplacian Function used to detect edges in an image [25].

$$\begin{pmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{pmatrix}$$

a)

$$\sigma^2(b) = w_1 \cdot w_2 \cdot (\mu_1 - \mu_2)^2$$

b)

Figure 22: a) represents the discrete approximation of the Laplacian function [25]. B) represents the Between Class Variance Formula which measures the separation between the edge class and non-edge class.

Figure 23: Laplacian Edge Detection applied on the Blurred Image

+1	0
0	-1

0	+1
-1	0

Figure 24: These rudimentary convolution masks provided a basis from which many methods of edge detection were developed. [26]

-1	0	+1
-2	0	+2
-1	0	+1

+1	+2	+1
0	0	0
-1	-2	-1

Figure 25*: These ma*sks emphasize the pixels that are nearer to the center of the grid, but they ignore the center pixels themselves. [26]

-1	0	+1
-1	0	+1
-1	0	+1

+1	+1	+1
0	0	0
-1	-1	-1

Figure 26*: These masks* provide a simpler approximation of the edges as they place the same emphasis on each of the surrounding pixels. [27]

2.3 Other Machine Learning Models and Methods

2.3.1 Classification and Regression

The two classes of supervised learning algorithms are classification and regression methods [28]. Regression models try to find the relationship between the independent variables (input)/output variables (output) to predict the continuous values. Figure 27 illustrates the differences between the two methods.

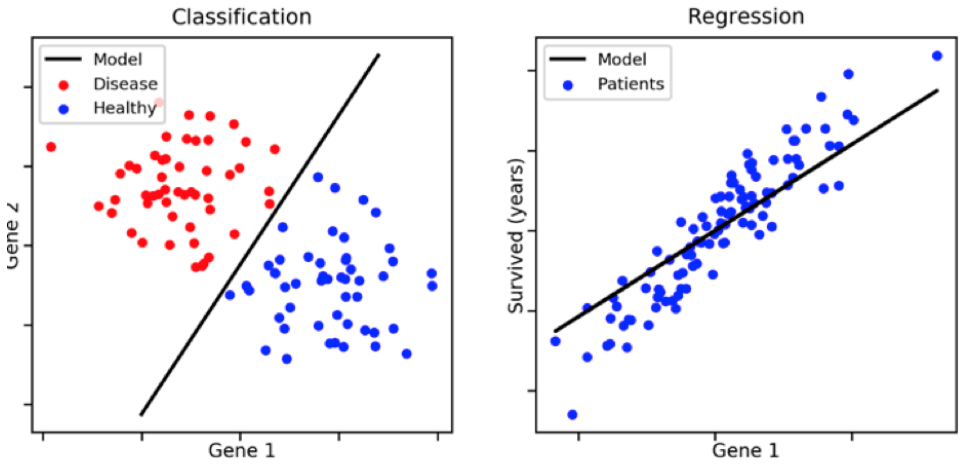


Figure 27: This figure illustrates the difference between classification and regression. Binary Classification is shown in the left image where the goal is to predict the probability of two pre-defined categories because it is binary level classification. The pre-defined categories are represented by discrete numbers (one and zero in this case). The categories in this example are Disease and Non-Healthy and we can see the linear line does its job of separating the two classes. Regression models predict continuous values which can be seen in the image on the right [28].

An example of classification in machine learning could be classifying a movie review as poor, good, and great [28]. The way the probabilities are calculated for each class poor, good, and great is through the SoftMax function shown in Figure 28. For

example, if the class scores were 5.0, 2.5, and 0.5, the probability for class Poor will be 0.9, 0.1 for class Good, and 0.0 for class Great.

$$Softmax(x_i) = \frac{(e^{x_i})}{\sum_{j=1}^{K}(e^{x_j})}$$

Figure 28: SoftMax Equation which assigns the Class Weight scores to assign a probability to a class [29].

Regression models predict a continuous output variable from the independent (input) variables [28]. For example, in this project we are using two consecutive semantic segmentation arrays to predict the change in the longitude, latitude, and altitude between the two arrays [28]. Another example of regression can be using square footage, land size, and city to predict the price of a house [28]. Some models used to perform classification/regression are tree passed models and neural networks.

2.3.2 K-Nearest Neighbors

Other classical methods of machine learning have existed for quite some time. The goal of this project was to develop something that will out-perform these older models. The two specific models that we looked at will be the k-nearest neighbors (KNN) algorithm and the Random Forest algorithm [30]. KNN works by taking a labeled dataset with known labels, and labeling a new data point based on which data is closest. Take the labeled dataset in Figure 29 for example.

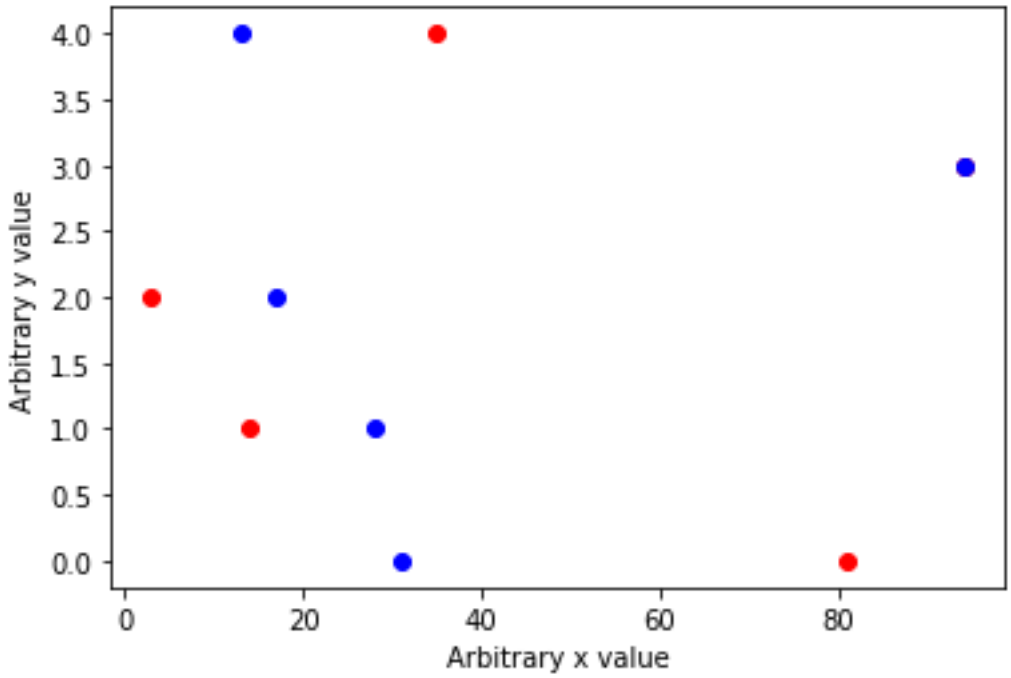


Figure 29: Example arbitrary data set before a new data point is evaluated by the KNN algorithm.

In this data set, some points have been labeled as blue, and some points have been labeled as red. Now, assume we have a new data point whose label we do not know.

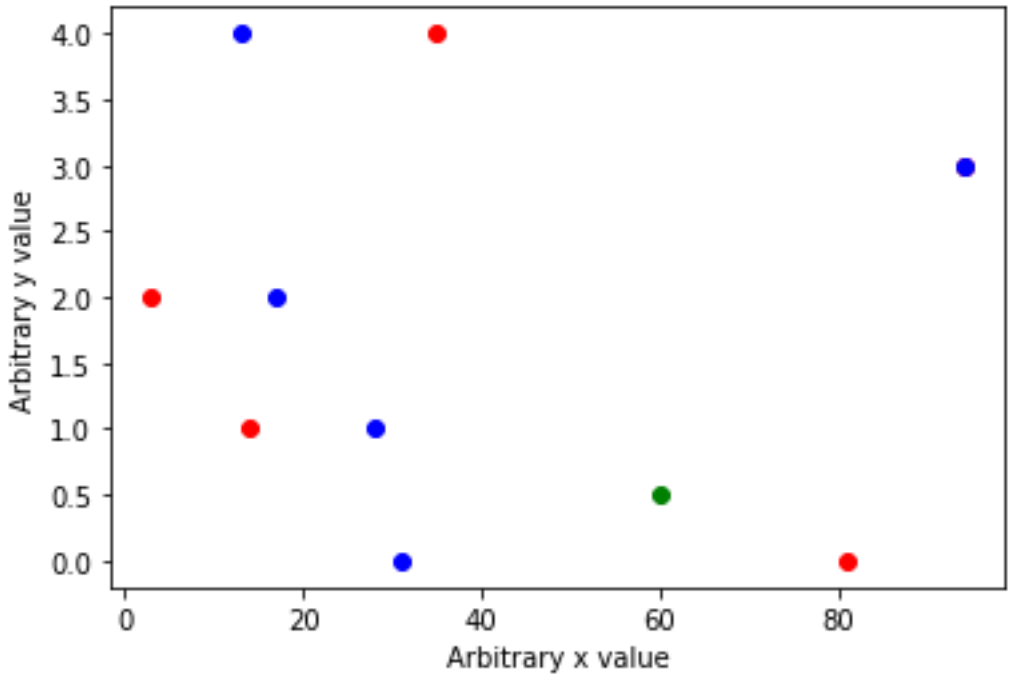


Figure 30: The same dataset as before, but now with a new data point (colored green) that we want to label as either "blue" or "red".

Here is where the "K" in KNN is introduced. We want to observe the labels of the new data point's "neighbors" (i.e., the datapoints that are closest). K is simply a metric of how many "neighbors" we observe. For example, in the figure above, the green point's closest "neighbor" is the red point at approximately (80, 0). Thus, with a value of k = 1, our new data point would be labeled as red to match its k-closest neighbor.

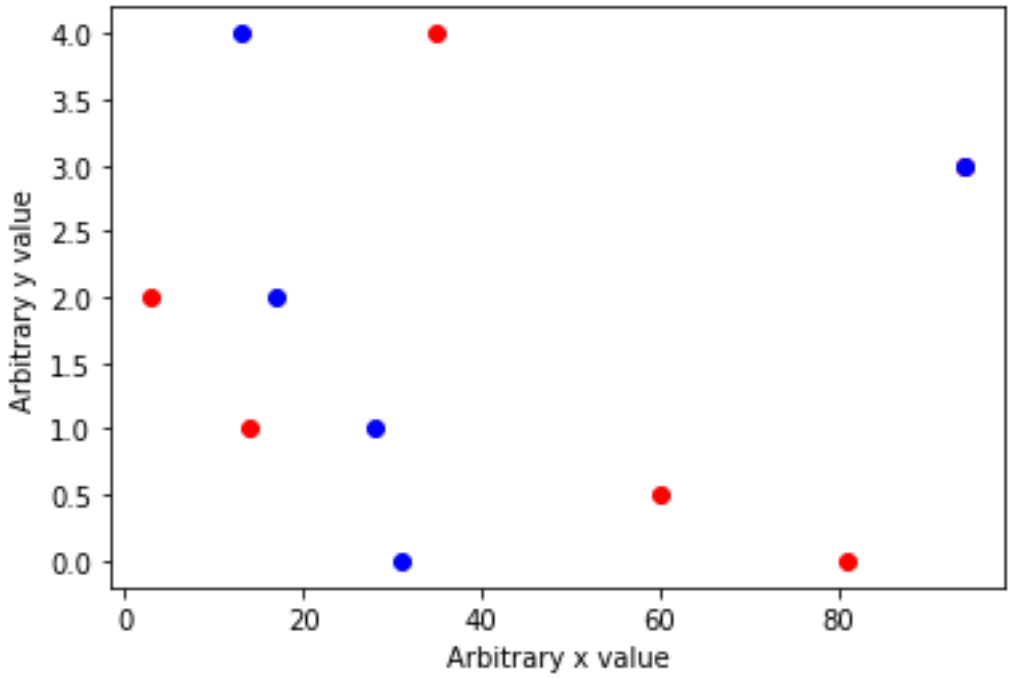


Figure 31: The prior data set with our new data point labeled as "red" with k = 1

However, if we would instead use a k = 3, we would then look at our new data point's 3 closest neighbors. These would be the same red point as before as well as the blue points at approximately (30,0) and (30, 1). Thus, our data point has more blue neighbors than red neighbors, and thus we would label our new data point as "blue."

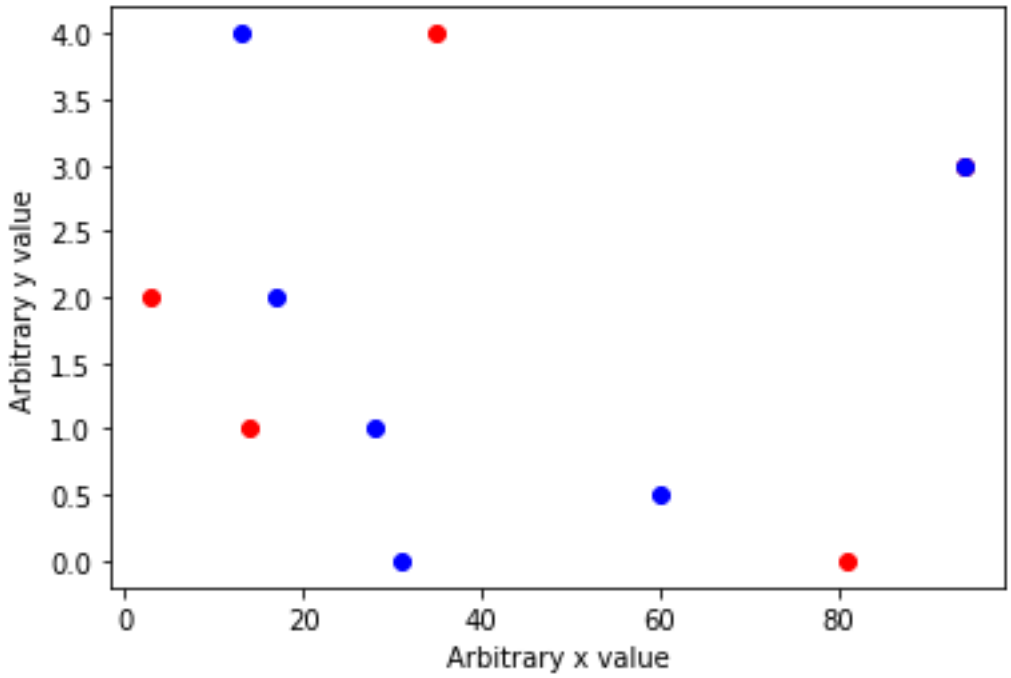


Figure 32: The same data set with our new data point labeled as "blue" with k = 3

The k value used for this algorithm can be tuned to minimize the prediction errors the algorithm may make. For example, say that we repeated this process for 1000 new data points. If you were to randomly pick a k value, the algorithm may misclassify many new data points. Thus, in practice we can test various k values on a training data set to select a k value with the lowest number of misclassifications.

2.3.3 Random Forest

To first understand a random forest, one also needs to also understand what a decision tree is. A decision tree is a simple algorithm to classify an instance based on its features [30]. Many people regularly use a rudimentary decision tree in their day to day

lives to make decisions without realizing it. A decision tree can decide which class an object is based on its features.

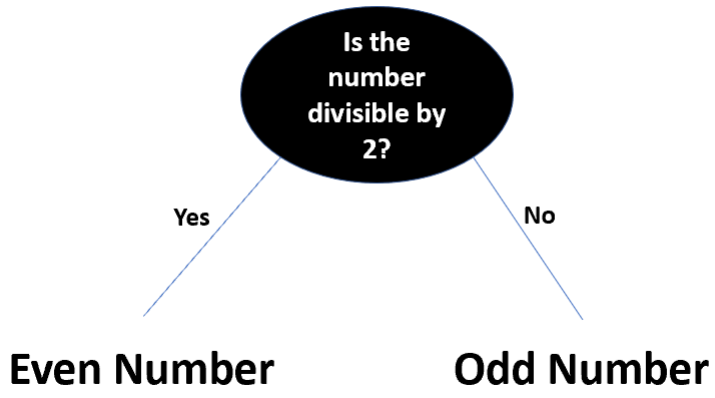


Figure 33: Example of a simple decision tree deciding if a number is even or odd.

As seen in Figure 33 above, a decision tree can decide what class an input instance is based on its features. In this specific case, that feature may be a number's divisibility, but in another case, there may be many more features. This variability of features is where the power of random forests can come into play. A singular decision tree may be limited by any number of factors, such as overfitting or underfitting [30]. However, a random forest takes many different trees, all using different subsets of the input data. These subsets use different combinations of features of the dataset rather than every single one. Thus, once every tree in our random forest makes a decision, we can treat each tree's classification as a vote with the most votes deciding our input's classification. With this method we can eliminate the shortcomings of one decision tree by itself.

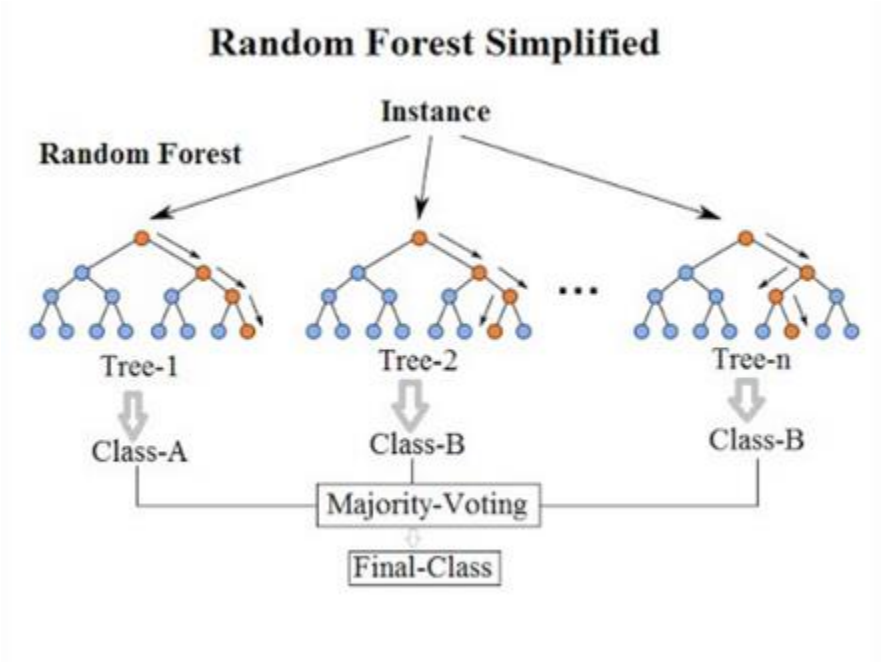


Figure 34: Example of a random forest classifier [31]

2.3.4 Neural Networks

Deep learning is a useful tool for semantic segmentation, as it can train a network to map between a semantic label and its visual appearances. There are various architectures that can be used for this, including VGG [32], which uses three-by-three convolution filters and between fifteen and twenty layers, and was found to perform well in localization and classification. ReNet is another architecture with a different approach [33], it replaces convolutional layers with recurrent neural networks (RNNs). ReSeg is a semantic segmentation method based on ReNet. There is also a deep residual neural network called ResNet that is useful for visual recognition [34]. It models the residual representation with a convolutional neural network (CNN). This convolution makes it much easier to train deep neural networks.

Neural networks are a type of machine learning algorithm which is inspired by the structure and function of the human brain [3]. The network consists of interconnected neurons which transmit information through layers of connections. Neural networks are commonly used for tasks such as Natural Language processing, semantic segmentation, speech, and image recognition [3]. The concept of neural networks was introduced in the 1940s by Walter Pitts and Warren McCulloch [3]. The main idea behind neural networks is to approximate the function $y = f^*(x)$ by learning the θ value in the function $y = f(x;\ \theta)$ to achieve the best approximation. Neural networks are networks because they compose of many different functions together. For example, there could be a function $f(x) = f^{(3)}\big(f^{(2)}\big)(f^{(1)})(x)))$. The $f^{(1)}$ function represents the first layer of the network which receives the input data. The $f^{(2)}$ function

represents the hidden layer and the $f^{(3)}$ represents the output layer. For each input x, the learning algorithm must decide how to construct the relationship between the hidden layers and input layers to best approximate $\hat{y}$. To learn and represent non-linear functions of x we can apply a non-linear transformation on an input x: $\phi(x)$ where ϕ represents a non-linear function. We can see an example representation of a neural network in Figure 36b. In image b) in Figure 35, the formula used to calculate the output for a hidden layer $x_0 * w_1 + x_1 * w_2 + b_1$ where b_1 is a bias term and w_1, w_2 are the weights which are randomly assigned at the start of the neural network training process [35]. These weights are altered during the training process to minimize/maximize the loss function depending on the scenario. The outputs of the hidden units are used with the weights to determine the final value of the output layer. The matrix representation of calculating the output of a neural network can be seen below:

$$X_{(1,n)=}[x_0, x_1], \qquad W^{(1)} = [w_1, w_2, w_3, w_4],$$

$$W^{(2)} = [w_5, w_6].\ b^{(1)} = [b_1, b_2]$$

The output in Figure 35 image c) can be calculated using the formula:

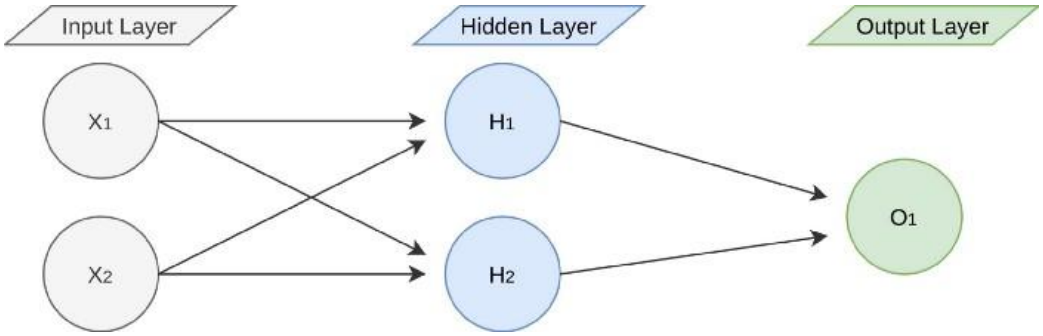
$$\big((x * w_1 + x_1 * w_2 + b_1) * w_5\big) + \big((x_0 * w_3 + x_1 * w_4 + b_2) * w_6\big) = \hat{y}$$

Which can be simplified to:

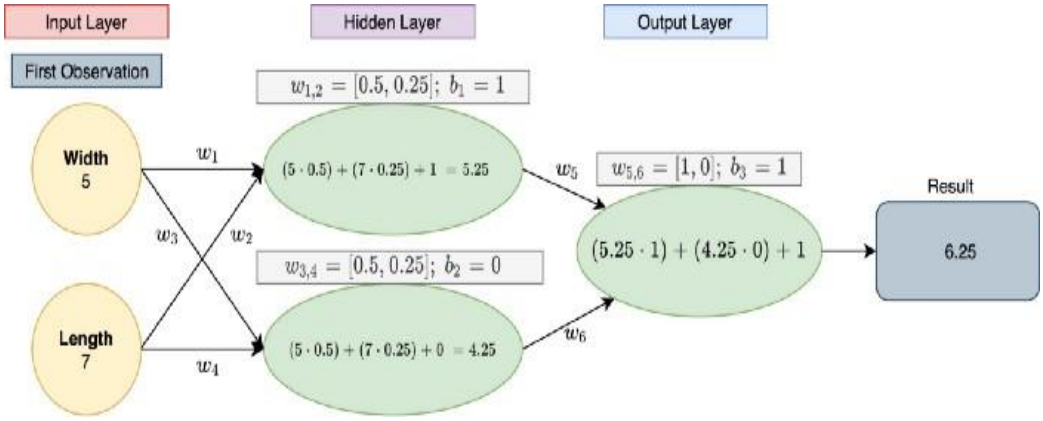
$$W^{(2)}(XW^{(1)} + b^{(1)}) + b^{(2)} = \hat{y}$$

Age	Width	Length
5	5	7
8	4	9
9	6	12
4	3	6
3	2	3

a)



b)



c)

Figure 35: Image b) represents a sample neural network architecture with an input layer, a hidden layer which passes the weighted sum through a non-linear function to

predict an output. In image c) we can see the width and length features are the first two neurons in the input layer [35].

2.3.4.1 Activation Functions

Activation functions in neural networks are mathematical functions which are used to introduce non-linearity to the output of a neuron allowing the model to understand complex relationships between the input and output [36]. The activation function applies a non-linear transformation to the weighted sum outlined in Figure 35 image c). This can be further illustrated in Figure 36 where weighted sum is calculated from the inputs and weights. The weighted sum is then passed into an activation function to produce an output. Activation functions also control the output range, for example, the sigmoid activation function is widely used for binary classification because a probability between 0 and 1 is applied to each of the classes. Activation functions also allow for some information to not be passed on to following layers depending on the threshold. Common activation functions include Sigmoid, ReLU, SoftMax, and Tanh functions [36].

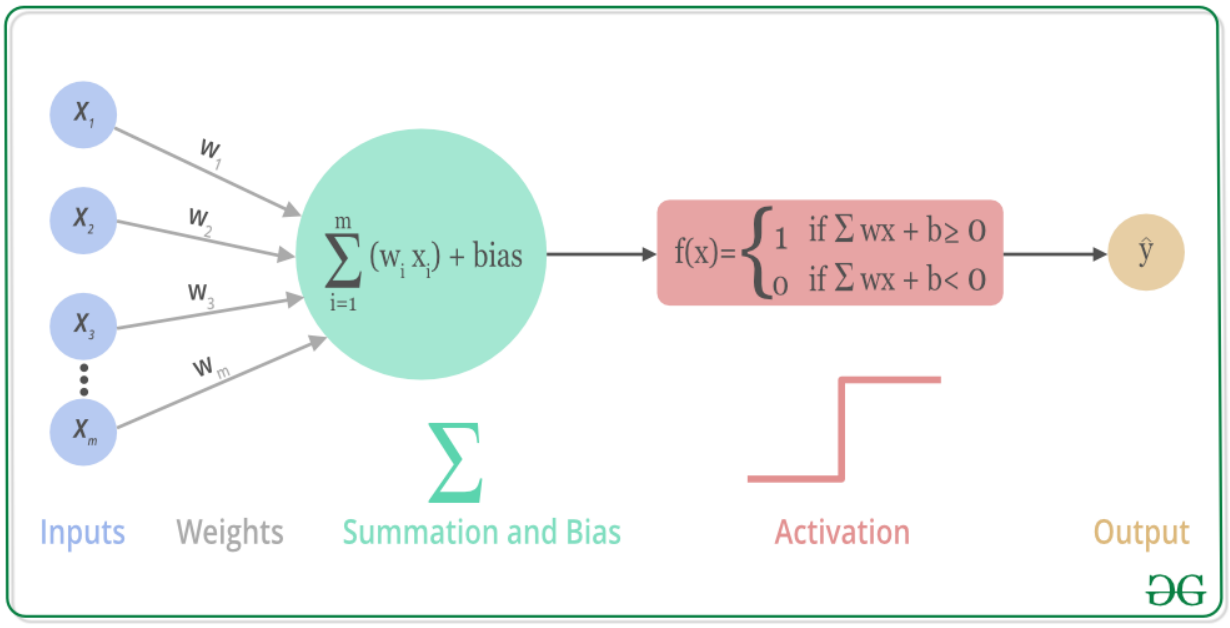


Figure 36: Weighted sum passed into activation function to produce an output. The activation function allows for a non-linear prediction model and to decide if information from previous layers should be passed onto future layers. [37]

The Rectified Linear Unit (ReLU) activation function is popular especially for transforming outputs in the hidden layers [36]. This is because ReLU is computationally efficient because negative values will be set to zero which reduces the complexity of the algorithm. ReLU looks to mitigate the vanishing gradient problem which stems from weights being very small and the training process becomes very slow. The ReLU function allows the derivatives to be either 0 or 1, so the gradients are calculated more easily throughout the layers of the neural network. However, a potential barrier for ReLU is the neurons that output zero from the ReLU function will not have their weights updated. The ReLU function can be illustrated in Figure 37.

$$ReLU(x) = \begin{cases} x & x \geq 0 \\ 0 & x < 0 \end{cases}$$

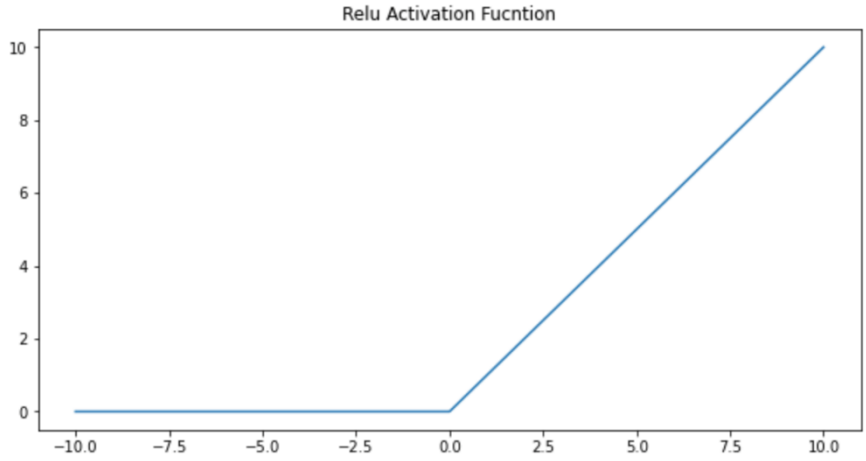


Figure 37: ReLU function graph. This function sets negative values to zero and values larger than zero will remain the same.

2.3.4.2 Optimizer/Training

Neural network optimizer functions are algorithms which are used to update the weights and biases of the neural network during the training process to minimize the loss function (function which measures the difference between the true and predicted output). Popular optimizers include Mini-Batch Gradient Descent, Adam, Adagrad, and RMSprop [38].

Gradient descent is a common optimization algorithm which forms the basis of Neural Networks. The term gradient descent stems from reaching the lowest point on the curve, for example, in an upwards opening parabola function we will look for the lowest point on that curve. The goal of optimization is to minimize the loss function which measures the difference between the actual and predicted outputs [38]. This is done by iteratively adjusting the weights and biases of the neural network. This is a

similar concept to linear regression where the coefficients are assigned so the sum of squared residuals: $SSR = \sum_i (y_i - f(x_i))^2$ is minimized. In classification problems, a common approach is to minimize the cross-entropy function which measures the difference between the predicted and true probability distributions of the target variable. The formula is as follows: $H = -\sum_i \big(y_i * \log\big(p(x_i)\big)\big) + (1 - y_i) * \log\,(1 - p(x_i)))$. The first step to train a neural network is to initialize the weights of the model randomly and set the bias term to zero [36]. The Batch gradient descent algorithm is when all the data is inserted into the algorithm at once. This version of the gradient descent algorithm is problematic because there is a risk of getting stuck during the gradient calculations. The gradient is calculated using the whole dataset, so the function will be minimized early. Mini-Batch Stochastic gradient descent picks random instances of the training data and computes the gradient which makes the gradient calculation much quicker with a lower amount of data points to account for. Regular Stochastic gradient descent calculates the gradient for one single data point per iteration, this is very time exhaustive, so mini batch looks to overcome this issue by allowing an arbitrary batch size number. An epoch is the number of passes where all the data is passed through the neural network. For example, if the Neural Network is trained on ten epochs in mini-batch gradient descent, there will be N weight updates per epoch where n represents the number of training samples. After calculating the initial outputs with the initial weights, the loss function is calculated to assess the initial progress of the neural network. The gradient is then calculated to minimize the loss function. Back-Propagation is then performed where the derivatives are calculated of the loss function with respect to the parameters of the output layer. The chain rule is then applied all the way back to the

first layer so the derivatives of the loss function will be calculated with respect to the first layer. After all the gradients are calculated, the weights and biases will be updated using the following formula:

$$W_{x_f} = W_{x_i} - a(\frac{\partial Error}{\partial W_x})$$

where W_{x_i} is the old weight and W_{x_f} is the new weight. The formula is calculated by subtracting the old weight value by the gradient error calculation multiplied by a learning rate value which determines how much the weights should be adjusted during each iteration in one epoch. Figure 38 illustrates the impact of the learning rate. This figure shows when the learning rate is too high, the training can be instable where there may be no convergence at a minimum point. When the learning rate is too low it may take too long to converge at a global minimum point where the optimizer can get stuck at a local minimum. The epoch value should also not be too large because a large value can cause the model to overfit because it has seen the sample samples many occurrences.

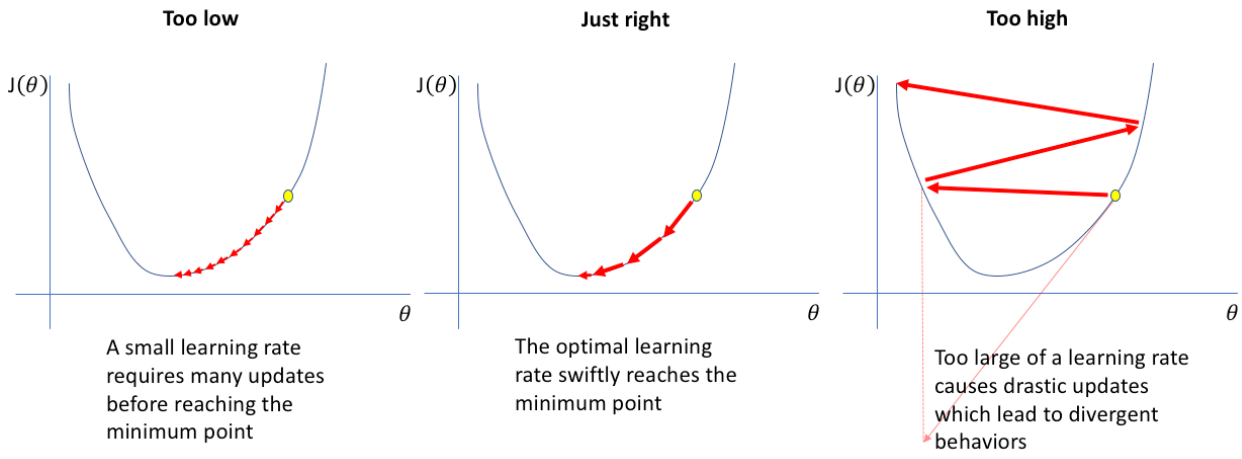


Figure 38: How Learning Rate Effects Performance [39]

Other variations of optimizers include RMSprop and Adam [38]. The RMSprop uses the gradient for optimization and an adaptive learning rate which changes over time. The learning rate is scaled by the moving average of the squared gradients which is divided by the current gradient. The scaling factor adjusts the learning rate according to the history of the gradients. The formula used to adaptively change the learning rate is as follows:

$$s = \alpha s + (1 - \alpha) * \nabla\theta\, J(\theta)^2$$

$$\theta = \theta - \frac{\eta}{\sqrt{s + \epsilon}} * \nabla\theta\, J(\theta)$$

Where s represents the moving average of squared gradients which is used to adaptively change the learning rate. Alpha is the decay rate which is a tuning parameter to specify the amount of change in the learning rate. η represents the learning rate and $\nabla\theta\, J(\theta)$ is the gradient of the loss function with respect to the weight parameter.

2.3.5 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are a type of neural network which is usually used for image/video processing tasks such as image classification, object detection, and semantic segmentation [40]. CNNs are designed to process grid-like data such as images. The purpose of convolutional neural networks is to process image data with the use of convolutional filter kernels which extracts/compress the useful features from a high-dimensional input [41]. Normal neural networks can be problematic because of the large number of parameters because of the fully connected structure; this makes the calculations computationally intensive when dealing with high-dimensional data. Normal neural networks also have trouble in learning a diverse set of features from the input images. Figure 39 shows the typical architecture of a CNN. The first stage of a CNN is using a convolutional layer filter to learn the initial feature map and to capture the different patterns in the image. Figure 40 shows how the convolutional filter is applied to compress the data to a smaller feature map. A 3x3 kernel is multiplied by sliding over each 3x3 area in the image to produce the output image. For example, the calculation for the last entry highlighted in yellow in Figure 40 is as follows:

$$(1 * 1) + (0 * 0) + (0 * 1) + (0 * 0) + (1 * 0) + (0 * 0) + (3 * 0) + (0 * 1) + (0 * 0) = 1.$$

The Pooling layer is used to further down sample the feature map derived from the convolutional filters into a lower dimension by still preserving the important features in the image [41]. Figure 41 shows a max pooling layer which takes the

maximum value in each of the 2x2 windows in the image to result in a 2x2 matrix. A common CNN architecture uses a Convolutional Filter, a pooling filter, another convolutional filter, and a final pooling layer which is flattened and used as an input into a typical fully connected neural network layer. After the pooling layer is inserted into the fully connected layer, the neural network computations are performed by the typical optimization procedures [41].

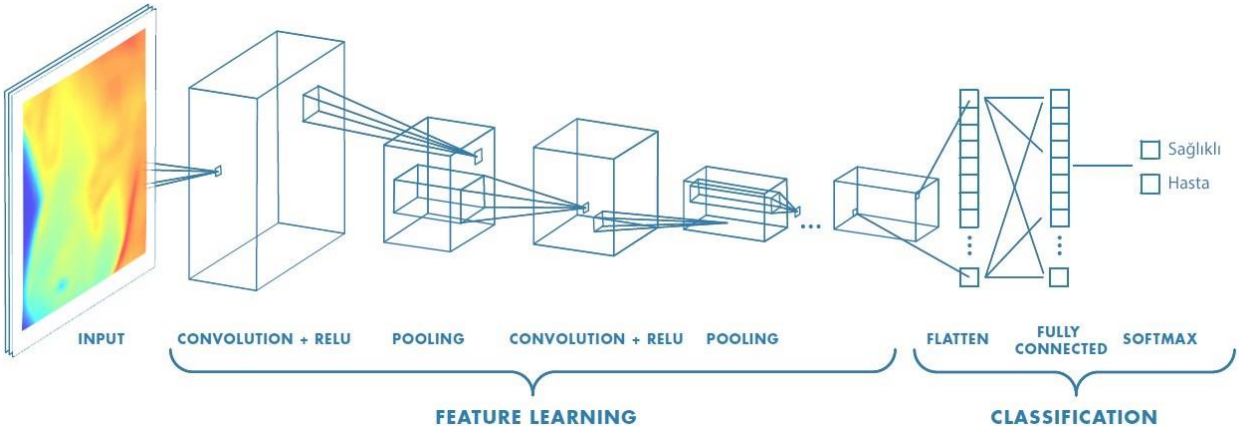


Figure 39: Typical Convolutional Network Architecture [42]

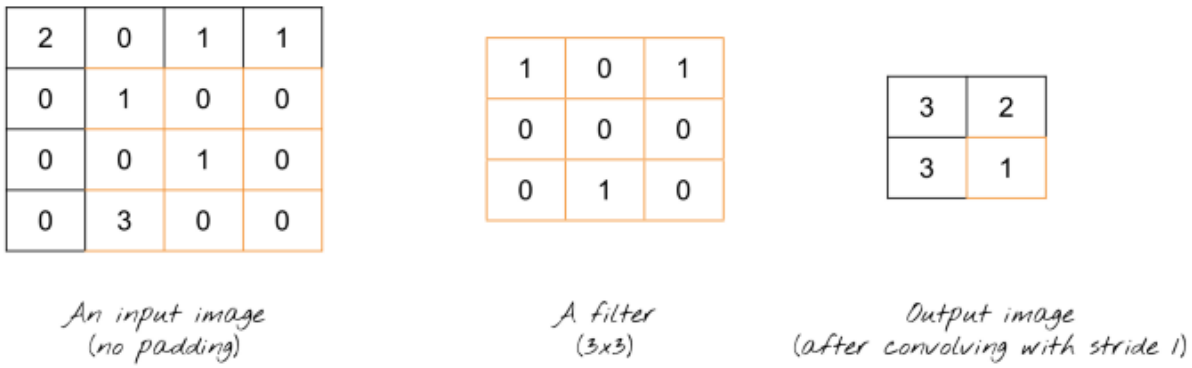


Figure 40: 3x3 Convolutional Filter Example [42]

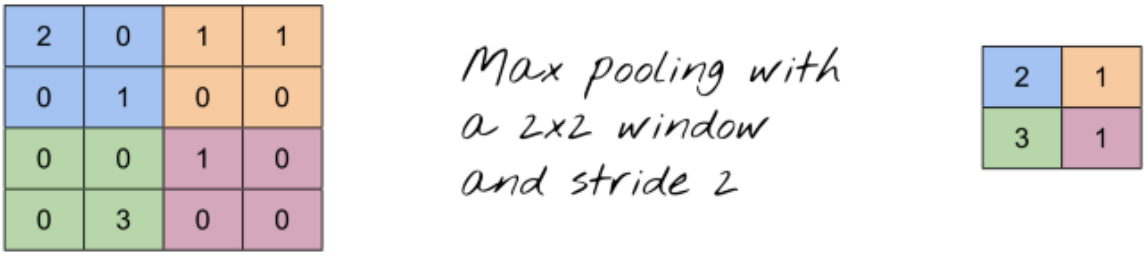


Figure 41: Pooling Layer [42]

2.3.6 Computer Vision Models

2.3.6.1 Semantic Segmentation

Semantic Segmentation is a computer vision task which involves dividing an image into multiple segments/regions [43]. Each pixel in the image is assigned a classification label which represents the region the pixel belongs to. In other words, Semantic Segmentation is pixel-wise classification. Figure 43 shows an example of segmentation mapping. The left image represents the original image, the middle image represents the predicted regions in the image, with purple representing the sky, yellow representing the sand, green representing the water, and blue representing the grass. The image on the right of Figure 42, shows the predicted segmentation 2-D array with mostly zeros at the top and ones at the bottom. This is because the zeros represent the purple region, and the ones represent the blue region in the middle image. Semantic Segmentation is used heavily for Autonomous Driving to identify different objects in the vehicle's view so it can avoid them. Semantic Segmentation predictions are calculated with the use of Convolutional Neural Networks.

Figure 42: Example Semantic Segmentation Mapping

2.3.6.2 Transfer Learning

Transfer Learning is a technique where machine learning models are trained on several tasks and data points and the trained model is used as a starting point for training a model on new tasks/data points [2]. A common approach to perform transfer learning is to acquire a pre-trained neural network on Python Deep Learning libraries such as Huggingface and TensorFlow [44][45]. The pre-trained network can be fine-tuned by providing a small dataset which is specific to the new task that the model wants to learn. The pre-trained model already can perform general tasks but fine-tuning the model can result in the model being able to perform more niche tasks to a specific problem. Transfer learning has been used heavily for object detection, image classification, and semantic segmentation. The steps in Transfer learning can be seen in Figure 43. The first step is to gather a large public dataset to allow the machine learning model to learn a wide range of broad tasks. For example, a robot's algorithm would need to teach the robot how to walk, run, and wave their hands. The second step is to learn the features of the data and train the model. Once the model is trained, the knowledge gained from the first model will be transferred to the second model. If the robot wants to learn a niche task such as shooting a basketball, a small basketball

shooting dataset can be passed into the new model because the robot already has the knowledge to run and walk.

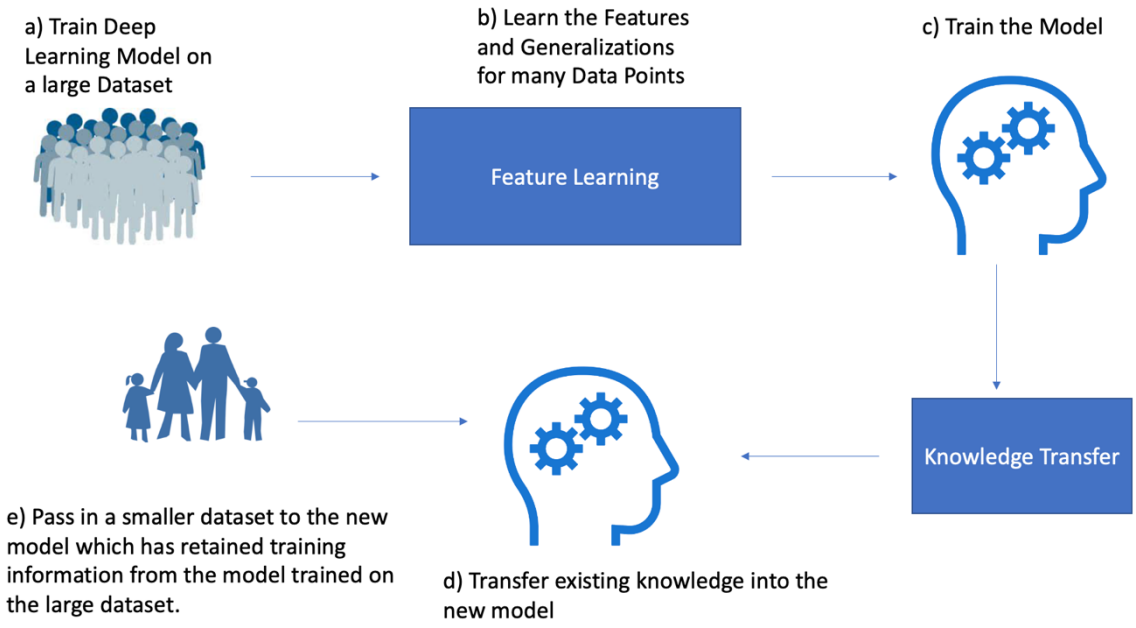


Figure 43: Transfer Learning Flowchart

2.3.7 Evaluation Metrics

Evaluation metrics are important to compare different models, hyperparameters and datasets. Common evaluation metrics for regression models are mean squared error (MSE) and root mean squared error (RMSE) [46]. Common metrics for semantic segmentation (pixel level classification) include the Jaccard Index and the Pixel Accuracy Score [46].

The mean squared error score represents the sum of the squared differences between the true value from the supervised learning dataset and the predicted value. It quantifies how different the predicted values are from the true data values. The equation

is shown in Figure 44. The Root Mean Squared error is the same equation as the MSE but the MSE equation will have the square root function applied to it.

$$\frac{1}{N}\sum_{i=1}^{n}(Y_i - \hat{y})^2$$

Figure 44: Mean Squared Error Equation

The pixel accuracy score shown in Figure 45 represents the percentage of correctly classified pixels versus the true pixel values from the semantic segmentation model.

$$Pixel\ Accuracy = \frac{(TP + TN)}{(TP + TN + FP + FN)}$$

Figure 45: Pixel Accuracy Evaluation metric for semantic segmentation.

Another popular evaluation metric for semantic segmentation is the Jaccard index shown in Figure 46 [47]. The calculation is Area of Intersection between the ground truth and the predicted semantic segmentation matrix divided by the union of the area between the ground truth and predicted semantic segmentation matrix.

$$J(A.B) = \frac{A \cap B}{A \cup B}$$

Figure 46: Semantic Segmentation Evaluation Metric Equation.

2.3.8 Segments AI

Segments AI was founded by Otto Debals and Bert De Brabandere in 2020. They created Segments AI to create a user-friendly tool to easily label computer vision data. Segments AI now provides a data labeling backbone to help streamline computer vision tasks such as semantic, panoptic, and instance segmentation. Figure 47 shows the interface for labeling data in Segments AI. The purple in the image represents the sky, the green represents the grass, the orange represents the sea, and the yellow represents the sand regions. Segments AI offers a few tools for data labeling. Two popular tools are the tracing tool where a user can trace around regions to shade a specific color and the paint brush tool where users paint the regions. Segments AI also offers the Superpixel tool illustrated in Figure 48, which gives you a suggestion on where different labels are in the image.

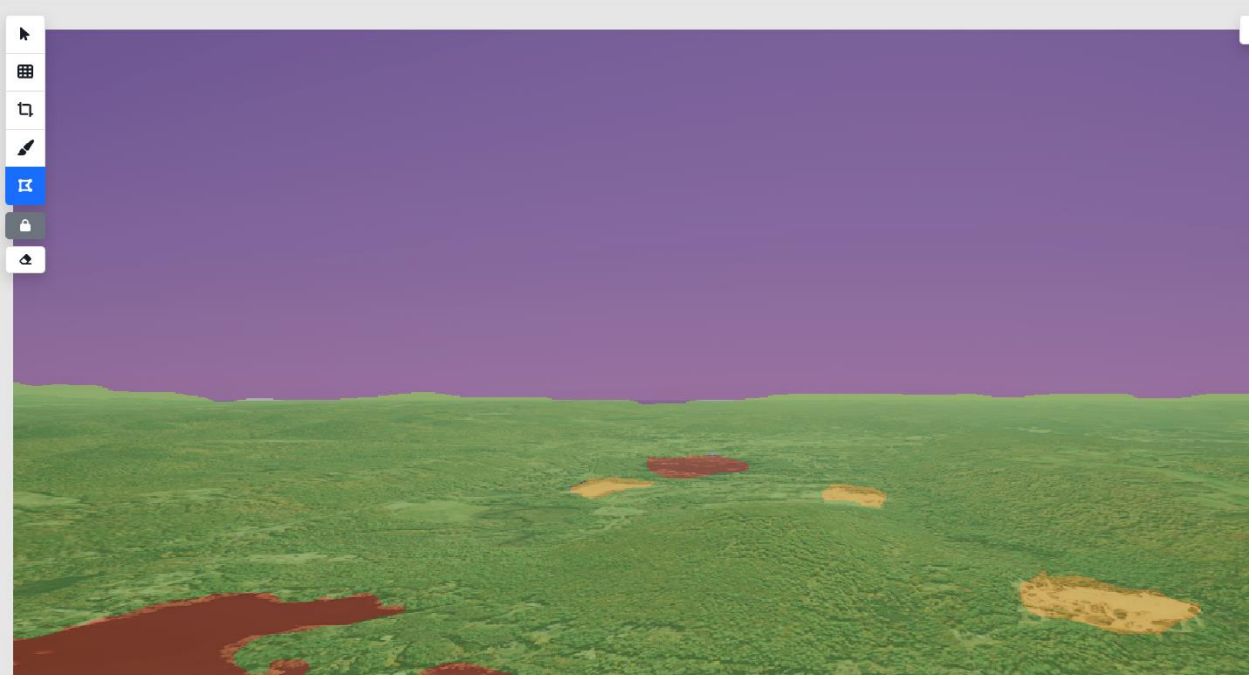


Figure 47: Example Segments AI Data Labeling

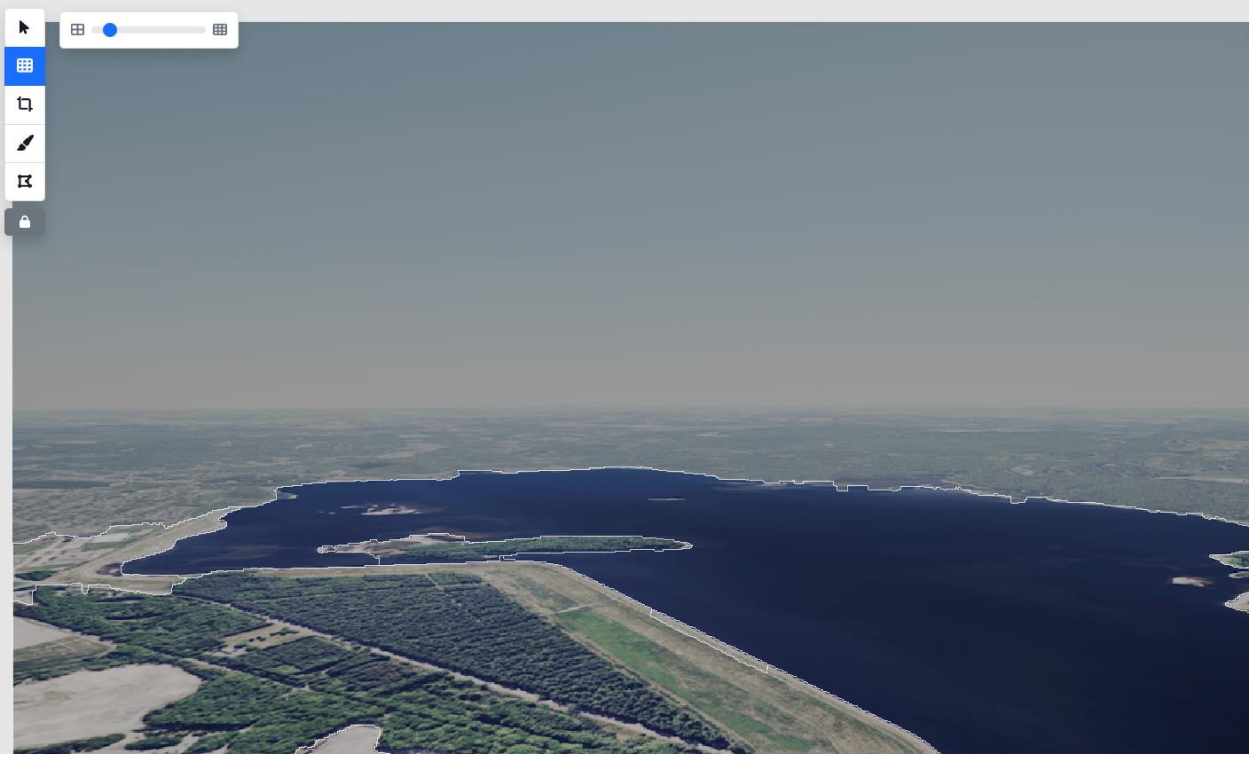


Figure 48: Superpixel Tool

3. Methodology

In this section, we will describe how we utilized the methods discussed in the background section to achieve the goals of this project. First, we will explain how we used Unreal Engine and Cesium to develop an intuitive user interface for the simulator. Then, we will describe how we created and improved our machine learning model for image recognition.

3.1. UI Development

The first goal of the project is to develop a user interface for the simulator that was built by Scheufele, Malabanti, and Spitaels. To begin this aspect of the project, we utilized the handbook that was written by Scheufele, Malabanti, and Spitaels. This allowed us to recreate their project in Unreal Engine that functions as a simulator for the parachute. This programming was completed using C++ and the Unreal Engine development suite. Scheufele, Malabanti, and Spitaels provided users with several parameters that they can control, including weather conditions, duration of the simulation, number of screenshots, time of day, time zone, flight path, and camera angle. These parameters could be changed, but only in the Unreal Engine development where the parameter inputs are hard to find and not user friendly [1].

We used a nested menu style of user interface with the main screen shown in Figure 49 and the settings screen shown in Figure 50.

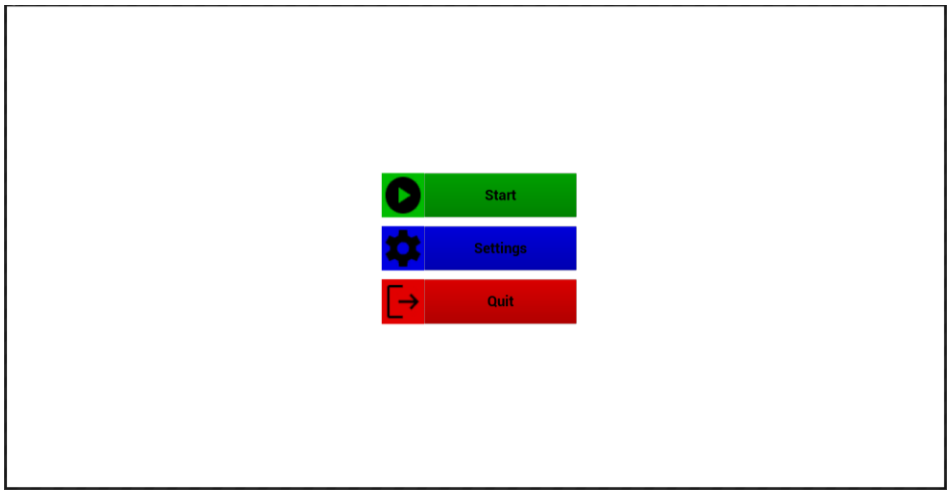


Figure 49: The main screen of the simulator.

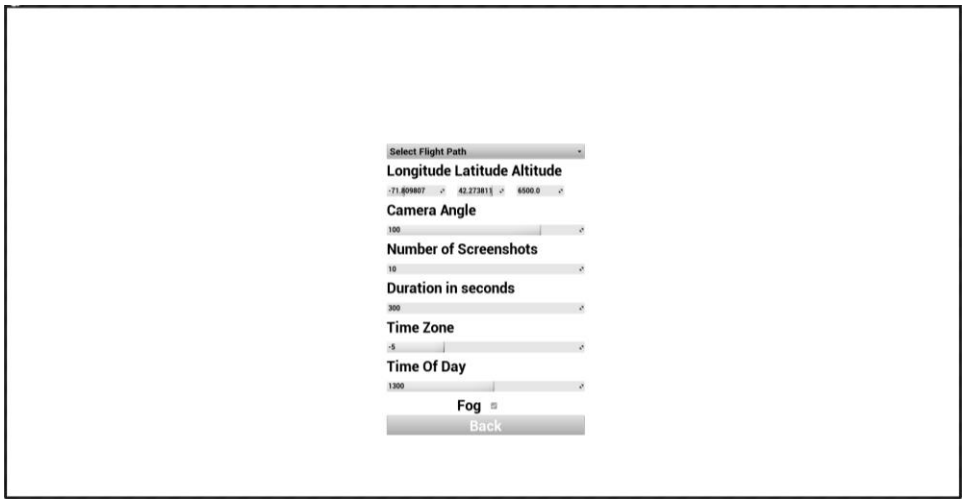


Figure 50: The settings screen of the simulator.

Figure 50 shows how the parameters that Scheufele, Malabanti, and Spitaels included could be input by a user into the simulator. The "Select Flight Path" dropdown menu allows users to select from 9 pre-installed flight path shapes. Based on Scheufele, Malabanti, and Spitaels' MATLAB code for flight path translation, we created a C++

function that will apply the flight path shape to a given starting location. This starting location can be input into the “Longitude”, “Latitude”, and “Altitude” number input boxes. The “camera angle”, “number of screenshots”, “duration in seconds”, “time zone”, and “time of day” all take numerical inputs as well. The fog parameter takes a boolean input in the form of a checkbox. These inputs are saved to a save file; the simulator then calls an event to apply the requested parameters.

3.2. Horizontal Flight Simulator

The horizontal flight simulator was made by taking two input locations then using the built in linear interpolation function of the Unreal Engine development suite [48] we can move the camera so that it travels from the input start to the input end. Many of the inputs from Scheufele, Malabanti, and Spitaels’ simulator were added to the horizontal flight simulator such as duration, number of screenshots, and time zone. The time of day is changed via a slider that was added from the Cesium plug-in [49].

3.3. Training Machine Learning Models

To train a model to be able to recognize the difference between water and land, as well as the difference between buildings and grass, we collected 108 images of New York City and manually colored them by these classes through a program called Segments AI [50]. Many of these photos did not contain much grass, so our main task was to differentiate between land and water. The images had varying levels of detail depending on their altitude, so for some of them we went as far as labeling where streets were located when possible. Images that were taken closer to the ground provided much

more detail. As such, we labeled these images with much more detail regarding the different classifications we were looking for.

When we train any type of machine learning model, it is important to measure the performance of the model. We can do this in a variety of ways by collecting metrics of the model when we try to utilize the model with test data. One of these metrics is the mean squared error (MSE) of a model [51]. MSE can be defined as the sum of the squares of the residual of every predicted data point divided by the number of total data points, where the residual is the difference between the actual data point and the predicted data point.

$$\frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2$$

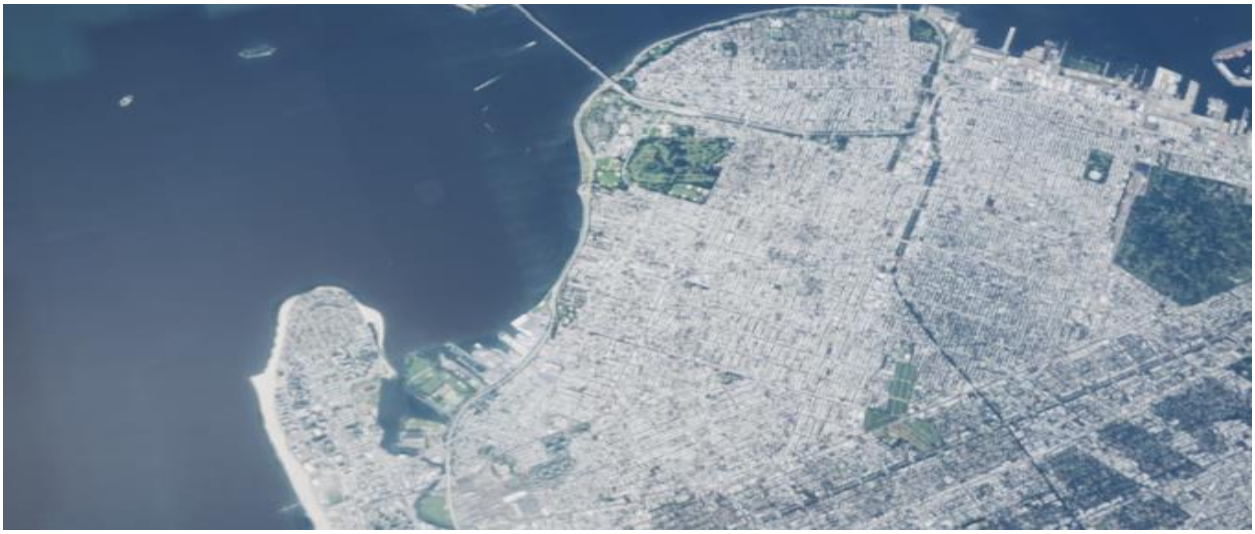
Figure 51: Mean Squared Error Formula, where n is the number of data points, Y_i is a true data point value, and $\hat{Y}_i$ is a predicted data point value [51].

3.4. Transfer Learning

3.4.1 Transfer Learning Dataset

Our first goal was to be able to detect the regions in the image. For example, in Figure 52a, we want to predict the sea area, building area, and grass areas. After being able to predict the regions in the image, our thought was to pass the predicted region location image from two consecutive images to predict the change in location between the two. Our thought process was that if the segmentation algorithm can detect the differences in the region that would be helpful in using a second image regression neural network to predict the location difference.

We wanted to train a model to be able to recognize the different regions in the image. The transfer learning model used was trained on more than 1,000,000 images from the simulators, so we collected a small dataset of 108 aerial images of New York City to learn the niche task in being able to detect regions from very large altitudes. 20/108 of the training data was used as a test set. The dataset was trained using a segmentation data labeler platform called SegmentsAI [52]. With the Segments AI platform, we were able to shade each region with a different color and this will serve as our 'Ground Truth' for a supervised dataset. In Figure 52, Image 52a will be the feature dataset (Independent Variable data) and Image 52b represents our true labels in the image (Dependent Variable Data). The goal of the Semantic Segmentation model is to receive the input image and predict the regions in the image as close to Image 52b.)



a)

b)

Figure 52: Image a) shows the original New York city aerial image. Image b) shows the labeled version of the image where orange represents the water, purple represents the buildings, and yellow represents the grass.

3.4.2 Transfer Learning Preprocessing

Our images from the flight simulator had a dimension size of 680x764x4. The reason there are four dimensions in this image is because it is an RGBA image. The first preprocessing step was to remove the black rectangles in the image to remove added noise in the image. The original image is shown in Figure 53. This technique was done by thresholding the black pixel values in the image and subtracting them from the image. To get more training data and obtain a more diverse training dataset, the color jitter and random rotation Data Augmentation tool in PyTorch was used. The images were also normalized where the pixel values are transformed to be between 0 and 1.

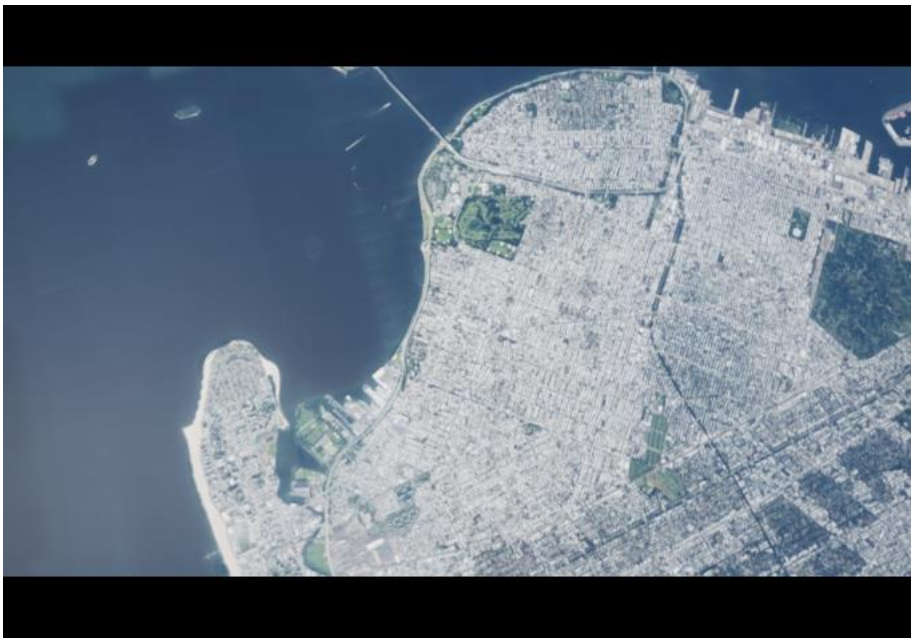


Figure 53: Original Aerial Image without Preprocessing

3.4.3 Transfer Learning Neural Network

The images were trained on a pre-trained "Nvidia/mit-b0" model on Hugging Face [44]. The hyperparameters of the model can be seen in Figure 54. This model was fine-tuned on a dataset of 88 training images and 20 test images. The Jaccard Index was used as the loss function for this task.

Batch Size	Epochs	Learning Rate
2	20	0.0003

Figure 54: Hyperparameters for Transfer Learning Model

3.4.4 Transfer Learning Error Metrics

The two metrics that we used to evaluate the semantic segmentation model are the Jaccard Index and the Accuracy score [46]. The Jaccard index is calculated by finding the overlap in the area between the prediction and ground truth divided by the total area. The pixel accuracy score is calculated by adding the number of True Negatives and True Positives and dividing that by the total number of pixels. These formulas can be seen in Figures 46 and 45, respectively.

3.5 Image Regression

3.5.1 Image Regression Dataset

The simulator produced an image location log which provided the longitude, latitude, and altitude for each image screenshot. Each of the 1100 cropped images from the New York City simulation were passed through the fine-tuned transfer learning model to produce a predicted segmentation image which can be seen in Figure 36. The goal was to pass two consecutive screenshots from the simulator to the transfer learning algorithm and insert the predicted segmentation image into the image regression neural network to predict the change in location. The image regression dataset can be seen in Figure 55. For example, the image paths from this figure will be passed into the transfer learning model to get the segmentation predictions. Since the first variation of the image regression problem was using images from the simulator which used a vertical movement, we decided to create a Convolutional Neural Network which predicts the

change in altitude because that the change in latitude and longitude is very small. The dataset included 1,100 image pairs, of which 25% were used only for the final validation process.

image1	image2	lon	lat	alt	lon2	lat2	alt2	lon_diff	lat_diff	alt_diff	coords_diff
HighresScreenshot00000.png	HighresScreenshot00001.png	-73.993	40.601	5899.966	-73.993	40.601	5892.386	0.000	0.0	-7.580	(0.0, 0.0, -7.579999999999927)
HighresScreenshot00001.png	HighresScreenshot00002.png	-73.993	40.601	5892.386	-73.993	40.601	5884.782	0.000	0.0	-7.604	(0.0, 0.0, -7.604000000000269)
HighresScreenshot00002.png	HighresScreenshot00003.png	-73.993	40.601	5884.782	-73.994	40.601	5876.796	-0.001	0.0	-7.986	(-0.0010000000000047748, 0.0, -7.985999999999876)
HighresScreenshot00003.png	HighresScreenshot00004.png	-73.994	40.601	5876.796	-73.994	40.601	5867.073	0.000	0.0	-9.723	(0.0, 0.0, -9.722999999999956)
HighresScreenshot00004.png	HighresScreenshot00005.png	-73.994	40.601	5867.073	-73.994	40.601	5857.432	0.000	0.0	-9.641	(0.0, 0.0, -9.641000000000531)

Figure 55: This table shows the dataset used to predict the change in location.

3.5.2 Image Regression Preprocessing

The original dimension of the image shown in Figure 53 is 680x764x3, there are three dimensions because the image is in RGB format. Some preprocessing methods which we looked at to compress our data were Canny Edge Detection, Sobel Edge Detection, and Laplacian Edge Detection. After the edge detection, the dimension of the image was 680x764x1. Pre-requisite steps for these edge detection algorithms include gray scaling and gaussian blurring. Our thought process behind this is that the Edge Detection data carries four times less data which further reduces the computational intensity. We attempted the image regression with and without edge detection.

3.5.3 Convolutional Image Regression Neural Network

Our team attempted to use a convolutional neural network to learn and compress the features from our high dimensional input images. We tried various CNN architectures by adding CNN and pooling layers. We first started with one Convolutional and Pooling layer, and we worked our way up to six layers. Each of the convolutional layers utilized the ReLU activation function. Figure 56 shows how the process of the feature learning by the output shape decreasing after being passed through each convolutional and pooling layer. The first layer has an input shape of 680x764x64 because the number of units chosen for the convolutional layer was 64.

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 680, 764, 64)	6464
max_pooling2d (MaxPooling2D)	(None, 226, 254, 64)	0
conv2d_1 (Conv2D)	(None, 226, 254, 64)	102464
max_pooling2d_1 (MaxPooling2D)	(None, 75, 84, 64)	0
conv2d_2 (Conv2D)	(None, 75, 84, 64)	102464
max_pooling2d_2 (MaxPooling2D)	(None, 25, 28, 64)	0

Figure 56: Shows how the output shape decreases after going through each convolutional and pooling layer.

The loss metric used for this network was the Mean Squared Error (MSE) loss function which measured the difference between the predicted and actual altitude from the image pairs [46].

3.5.4 Network Architecture

The architecture that we decided to go with was using six convolutional and five pooling layers. Smaller architectures weren't enough to adequately learn the full feature maps of the input images. We can see in Figure 57a the output size decreases every iteration. In the final layers of the network, the model compresses to a 1x128 flattened array which is used to produce the final output. Figure 57b shows the hyperparameters for the network. 50 Epochs, the Adam optimizer, and a learning rate of 0.001 was used to train this CNN architecture.

Layer	**Input Size**	**Output Size**
Input: CNN (5,5) Kernel 64 Units Max Pooling Layer (2,2) Window	**60x764x64**	**N/A**
Input: CNN (5,5) Kernel 64 Units	**60x764x64**	**226x254x64**

Max Pooling Layer (2,2) Window		
Input: CNN (5,5) Kernel 64 Units Max Pooling Layer (2,2) Window	**226x254x64**	**75x84x64**
Input: CNN (5,5) Kernel 64 Units Max Pooling Layer (2,2) Window	**75x84x64**	**25x28x64**
…	**…**	**…**
Flatten/Dense	**2x3x32**	**1x128**
Output	**1x128**	**1**

a)

Epochs	**Learning Rate**	**Optimizer**

50	**0.001**	**Adam**

b)

Figure 57: a) represents the architecture of the convolutional network. b) represents the hyperparameters used for the model

3.5.5 Error Metrics

The performance metric used to evaluate this model was the mean squared error (MSE) metric. This function measures the difference between the true and predicted values. The reason we didn't also use the RMSE which is the square root of the MSE is because our MSE values were already a small value. Our first baseline metric was testing our model's MSE vs the variance score. The variance score is simply the same equation as the MSE, but the mean of the dataset is substituted with the predicted output. So, the variance is the predicted mean for every data point. This equation can be found in Figure 44.

3.5.6 Additional Convolutional Architectures

Some additional architectures that were tested were testing different optimization functions such as AdaGrad and RMSProp [38]. We also attempted to test different numbers of convolutional/pooling layers to see how the performance was affected.

4. Results

This section details the results of our project. This includes a user interface that allows users to easily adjust the simulator as they would like with parameters such as starting location, duration of the drop in seconds, camera angle, time of day, time zone, number of screenshots to take, whether there is fog, and nine preinstalled flight paths. There are inputs available for each of these parameters which allows for a wider range of simulations. The other part of this project is the program we developed that can recognize regions and items in the simulation via a neural network that we developed. We will summarize the functionality of our new user interface and image recognition program.

4.1. Simulator User Interface Improvements

While the simulation developed by last year's team provides useful information on how a parafoil will travel through the air, it was not as user-friendly as it should be. Those using the simulator at DEVCOM would have a challenging time determining how to get the information that they want from the simulator without help from the team that created it. Our improvements allow these users to easily control the simulation and receive a clear output.

In any mission planning situation, the military needs some information to start with. The minimum information required to deliver something with a parachute is the weight of the load and the desired landing location [7]. While our simulator bases its location on the starting location, we provide information on this location and other

conditions. Other inputs that users can decide on include weather conditions, the time of day and time zone, and the starting location of the drop. The updated simulator also provides a drop-down menu of flight paths, which allows users to view some of the options for how the supplies may fall.

One issue that we found in our updates to the simulator was that when users change the location, the flight path was not adjusted accordingly. The default location for the simulator is in southwest Arizona, and minor adjustments (such as moving the starting location a few miles) worked, but we found that when we input a location further away, such as Worcester, MA, the curvature of the earth would not be accounted for. This led to the flight path not ending on the ground, but traveling almost horizontally through the air, as it did not rotate around the center of the earth to get to this new location from the default, but rather moved directly to the new location. We solved this problem with the help of last year's team, as we studied some of their MATLAB code that dealt with translations. Our solution was to change the code so that rather than basing a flight path on its starting and subsequent locations, the default state of the flight path is to start at [0,0,0] for latitude, longitude, and altitude. The movement of the parafoil through the flight path is then determined by the difference between the starting point and its current position. Making this change also solved another problem that we faced, where the coordinates of the flight path would change whenever the simulation was run, so if we ran the simulator in one location and then in another, the flight path would base its coordinates on the movement of the previous flight.

There is a problem with how the simulator takes inputs for latitude, longitude, and altitude. When users type in a coordinate with only one or two decimal places, the simulator will automatically adjust the input to a more precise number. This is not favorable, as these changes are arbitrary and can compromise the accuracy of the input. We also needed to determine how pausing the simulation will affect the screenshot functionality. Originally, we found that when we ran the simulation with the screenshots, if a user paused it, the screenshots would still be taken at the predetermined time, which would end up being the same picture repeatedly if it were paused for long. We solved this problem and pausing the simulation now also pauses the timer for the screenshots, so they will be taken at the desired interval.

While the handbook we were given to create the simulation was thorough, there were minor issues we faced while using it to develop our system. We originally endeavored to implement the simulation using Unreal Engine 5.0, while the handbook was written based on Unreal Engine 4.26. The differences between the two that we found included a different ordering of the steps that it takes to set up a project. We decided to download Unreal Engine 4.26 to use the handbook. There are also differences in some documentation used in the simulation coding.

4.2. Semantic Segmentation Datasets

We built a dataset to train and test a semantic segmentation model to predict the regions in the parachute image feed. This section will outline the methods and techniques used to train the Segmentation model.

4.2.1 Preprocessing

4.2.1.1 Resizing

The first preprocessing step we used was resizing our images from the flight simulator. An example image can be seen in figure 60. The images had black rectangles at the top and bottom which can add extra noise to the data. Removing these black rectangles can allow the model to differentiate the regions more easily because there is less data to process. The dimension of the image was reduced from 474x680x3 to 382x680x3 which overall lowers the computational complexity and speeds up the machine learning training process. The three represents the number of channels in the image and there are three because the image was converted from RGBA to the RGB format.

4.2.1.2 Random Rotation and Color Jitter

The second preprocessing step we used to prepare the data for the Semantic Segmentation model was the color jitter and random rotation augmentation. An example of the data augmentation can be seen in Figure 12. These are data augmentation techniques which generate more training data which diversifies the training set. With a

more diverse training set, the machine learning model can learn different scenarios which allows for better generalization of unseen data.

4.2.2 Semantic Segmentation Scenarios

We trained our Semantic Segmentation model on our overall dataset, but we also tried training the model for individual scenarios. The figures below outline the different scenarios in our dataset. In Figures 58 and 59, the regions in the image are easy to differentiate because of the varying color. But in Figure 60, it is even hard for the human eye to classify the regions because the road and the buildings are similar in color. The results for the different scenarios can be seen in Figure 61. The model was run four times on different shuffles of the train/test dataset.

In Figure 64, we can see the Jaccard Index scores for the Grass and Building scenario is around 0.98 with one outlier of 0.92. The labels in this image were grass and buildings. The results are good, and this is largely because the grass and building areas are distinct as the buildings are mostly white color, and the grass is green color.

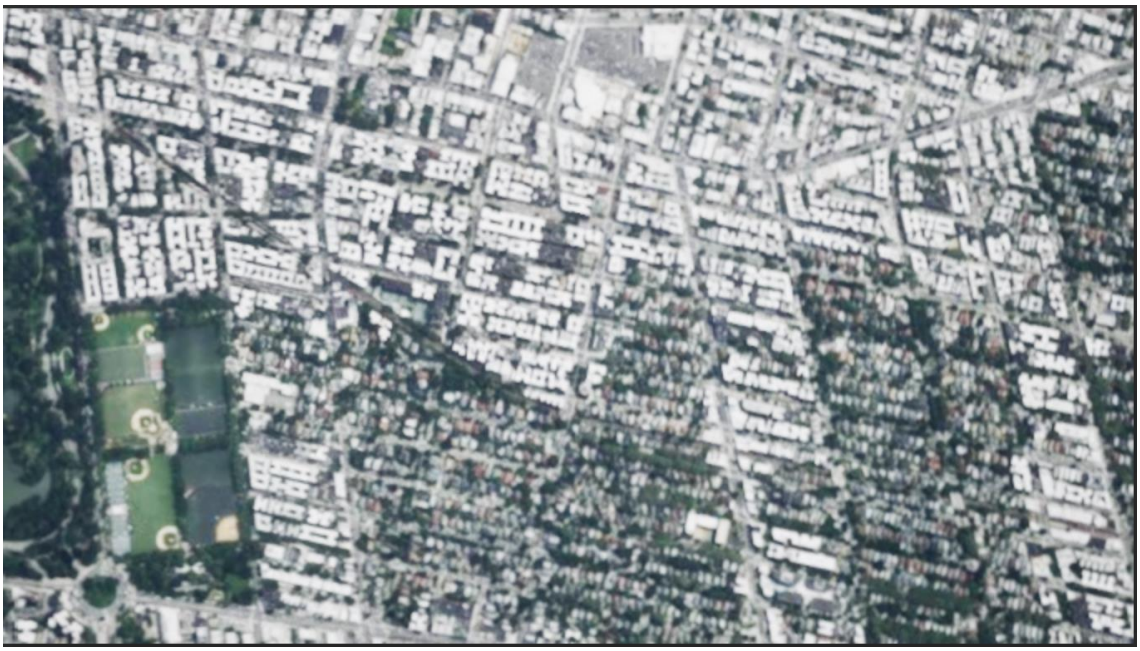


Figure 58: This figure represents the first scenario in our dataset where the image consists of mostly grass and buildings.

The building and sea scenario also performed well which can be seen in Figure 62. The average Jaccard Index score was around 0.90-0.91. The Sea and Buildings are large areas that can be separated easily.

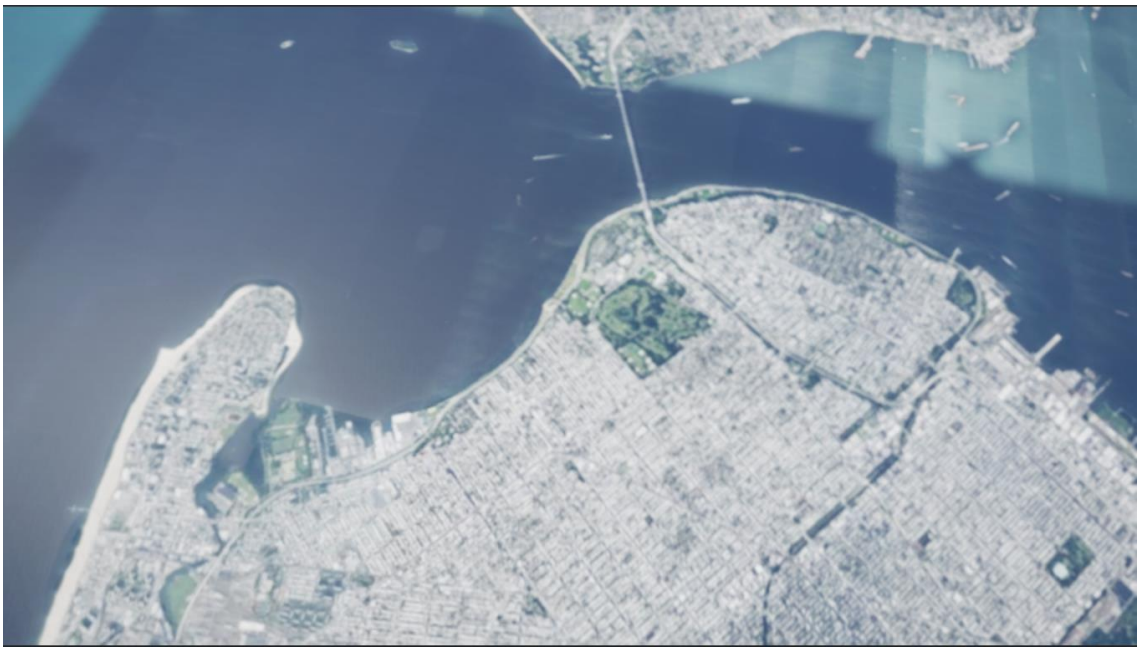


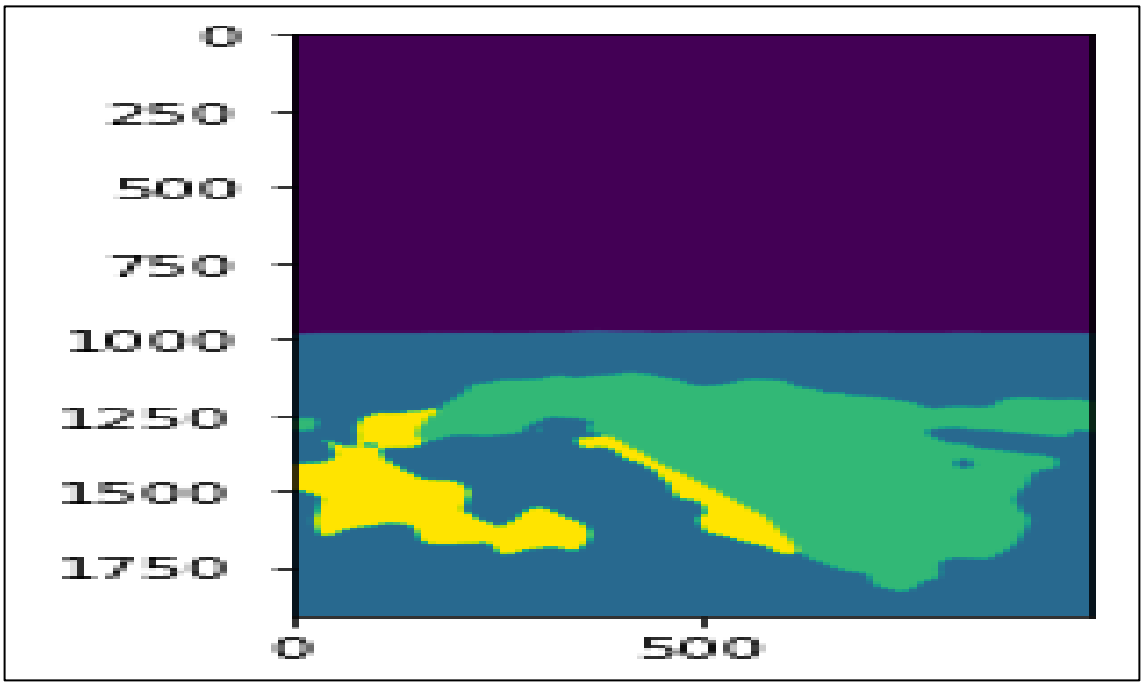
Figure 59: This figure represents the second scenario in our dataset which consists of buildings and the sea.

The results for the Road section were the worst out of all the scenarios as the Jaccard index was around 0.64-0.65. The labels for this scenario were the grass, road, and buildings. This is a difficult problem for even a human to solve because it is a high-altitude perspective with the buildings and road being similar in color.

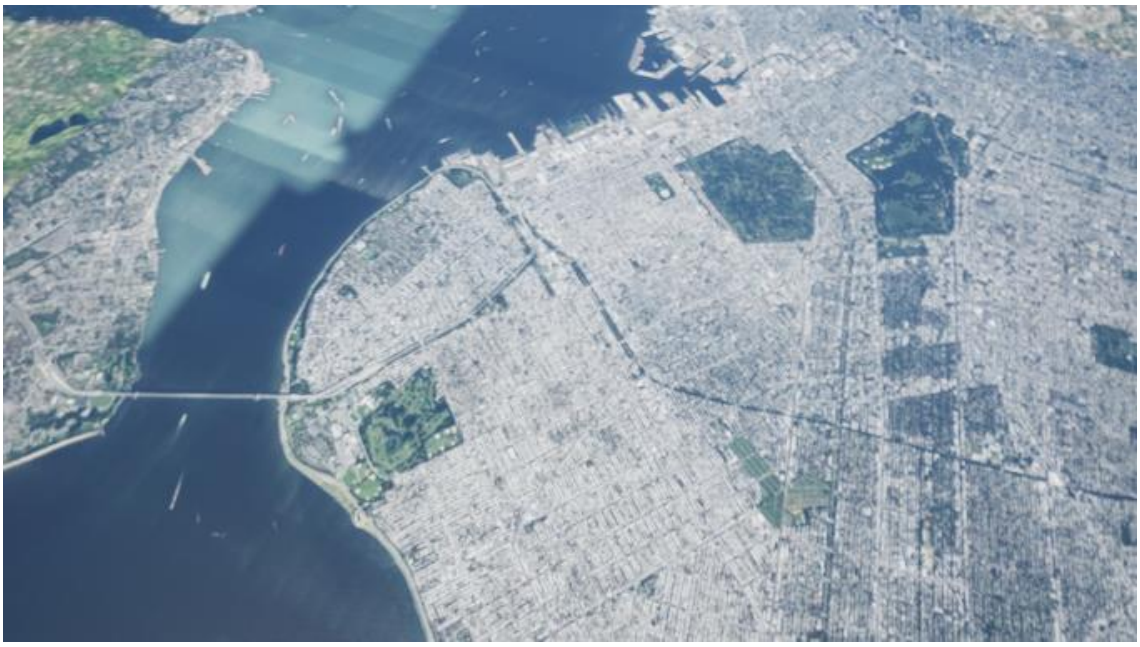
Figure 60: This figure represents the third scenario in the dataset which consists of road and building spaces.

The scenario in Figure 63a is the horizontal perspective of the images. We wanted to see how the model would perform on the horizontal perspective vs the vertical perspective. This model was trained with only 20 training images, and it performed well with around a 0.70 Jaccard Index score. The horizontal angle didn't need as much training data points to achieve a satisfactory result.

a)



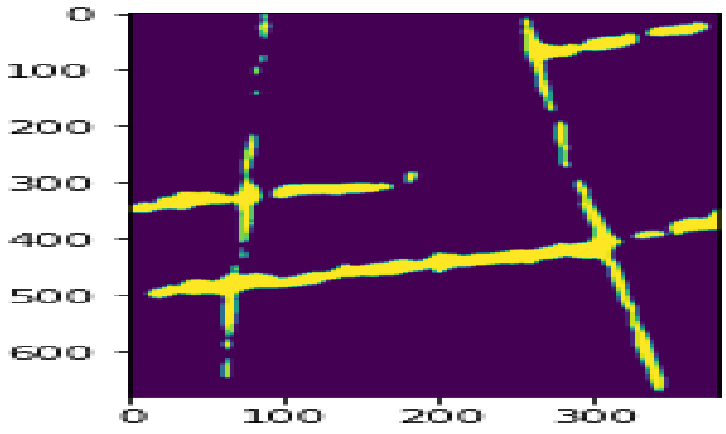
b)



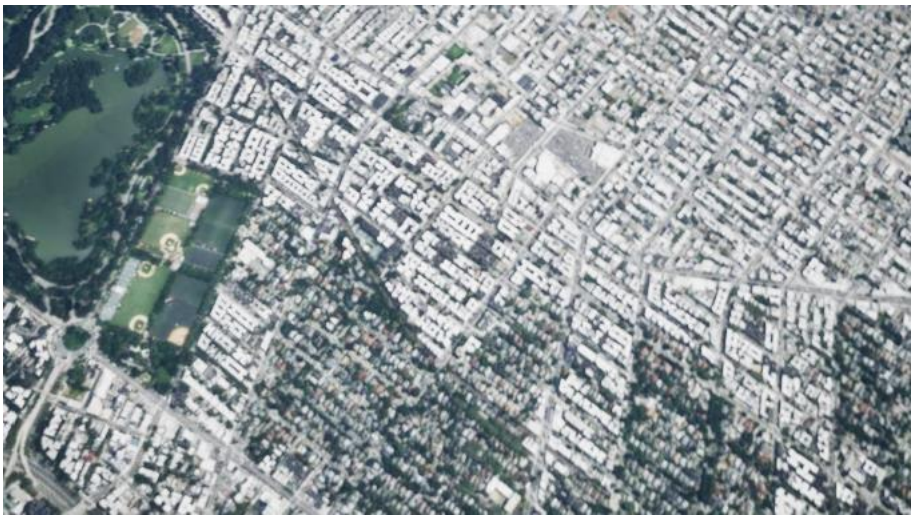
c)

d)

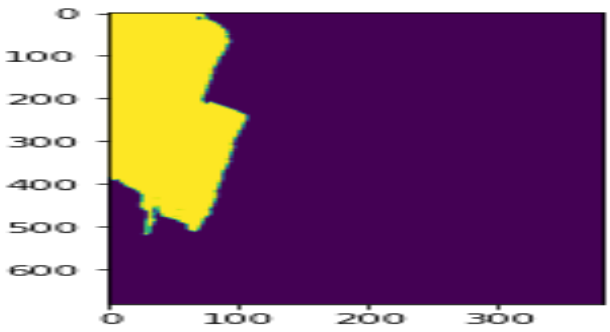
e)



f)



g)



h)

Figure 61: The images in a) and b) are the original image and the b) and d) represent the predictions from the Semantic Segmentation model.

In Figure 63, we can see that image b) is the prediction semantic segmentation result from using image a) as an input. The model was able to predict the sand, grass, water, and sky in the image.

Iteration	Grass-Building	Building-Sea	Horizontal	Road-Building	Overall
1	0.922918	0.922918	0.811685	0.648504	0.690477

2	0.980441	0.904813	0.880133	0.63857	0.709498
3	0.980107	0.917895	0.782596	0.644195	0.681446
4	0.980156	0.21733	0.815503	0.656089	0.712332

Figure 62: Accuracy of our model in differentiating between different classes of region.

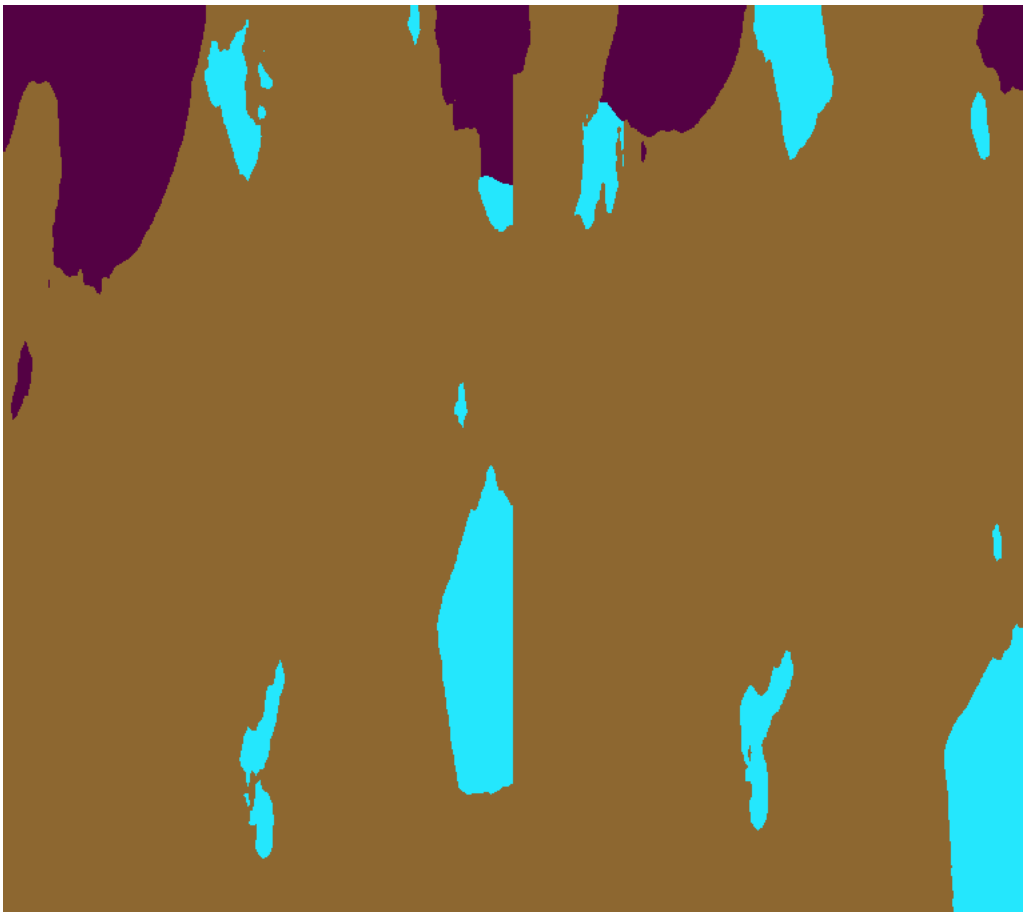
4.3. Image Regression Datasets

We utilized the trained semantic segmentation model to generate predictions for all 1,100 images from the New York City Flight Simulator data. The goal was to use two predicted segmentation arrays from consecutive frames in the video feed to predict the numerical altitude location change between the images. The idea behind this is, if one knows the starting location and the change between consecutive frames, they can calculate the final location.

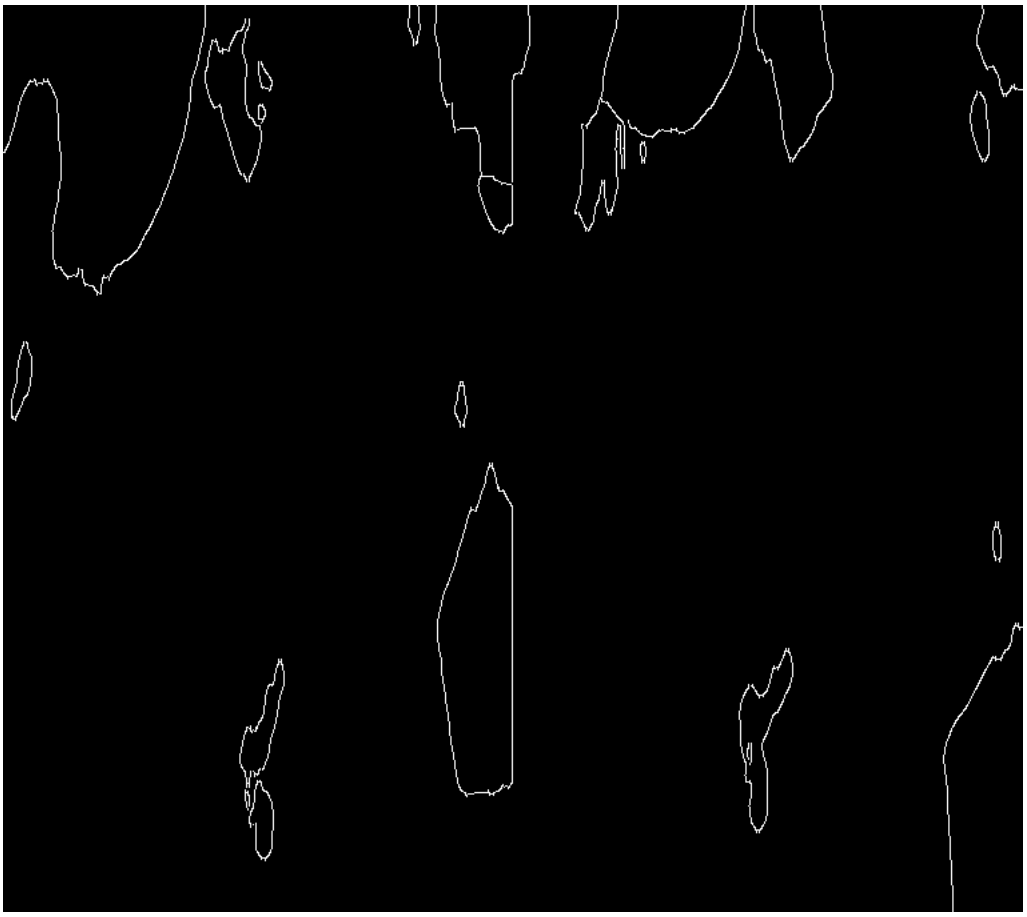
4.3.1 Preprocessing

We performed various pre-processing methods such as Laplacian, Canny, and Sobel Edge detection to attempt to compress the information to lower the computational intensity of the machine learning computations.

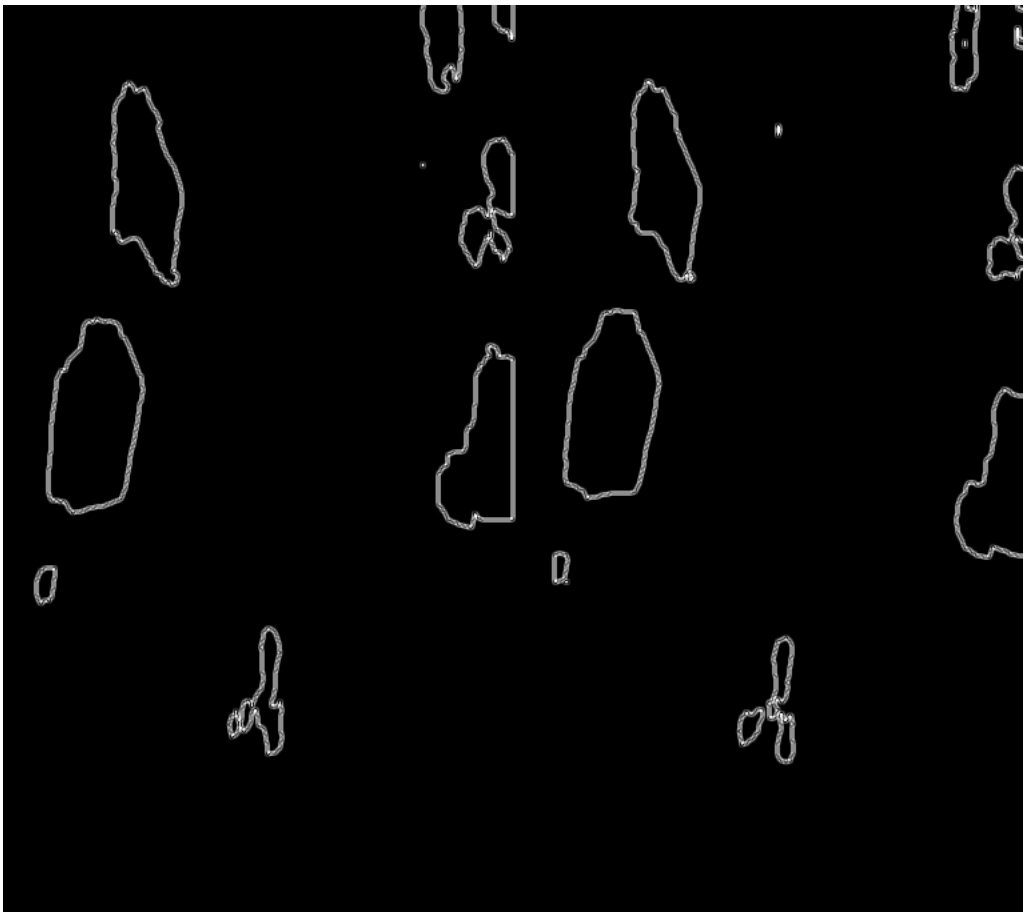
The image transformations can be seen in Figure 63. The semantic segmentation prediction was transformed with various edge detection methods. We also performed canny edge detection with the original image pair which can be seen in Figure 64.



a)



b)

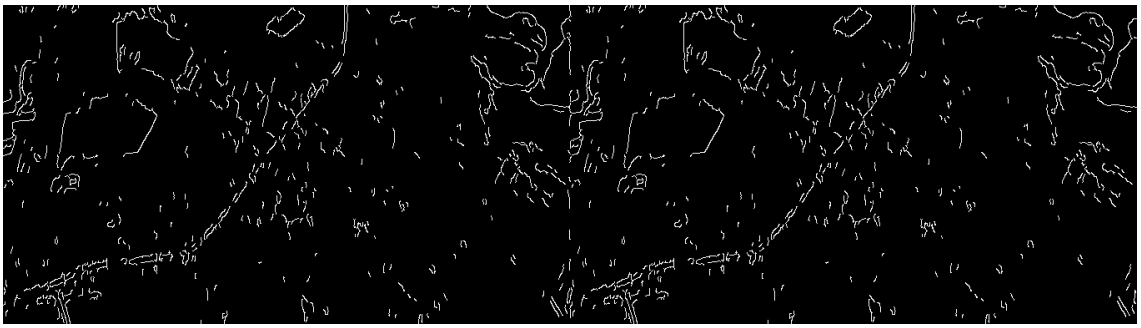


c)

d)

Figure 63: Image a) represents the original segmentation prediction from two consecutive images from the flight simulator. Image b) represents the canny edge detection transformed image, c) represents Laplacian Edge Detection, and d) represents Sobel Edge Detection

a)



b)

Figure 64: Canny Edge Detection performed on the original image pair. A semantic segmentation prediction was not used for this canny edge detection transformation.

Iteration	Canny	Laplacian	Sobel	No Edge	Original Image with Canny Edge
1	1.02	1.66	1.21	0.92	0.69
2	1.01	1.69	1.24	0.91	0.7
3	1.03	1.69	1.23	0.92	0.68
4	1.04	1.72	1.25	0.93	0.69

Figure 65: MSE Scores across four different train/test splits.

The results for predicting the altitude change with two consecutive image pairs can be seen in Figure 65. The scores are in terms of MSE and the original image (no semantic segmentation) with Canny Edge detection performed the best with an MSE of about 0.69. This score can be compared to the variance of the Altitude of the test set which is around 3.15. So, an MSE of 0.69 is significantly better. It is also important to note that the model trained on the semantic segmentation predictions performed better

than the model trained on the edge detection transformed semantic segmentation predictions.

The edge detection methods did not seem to play a significant role in lowering the MSE score; this is likely due to there being no color in the images. The color in the images played a huge role in differentiating the regions from each other which helped to predict the locations more easily.

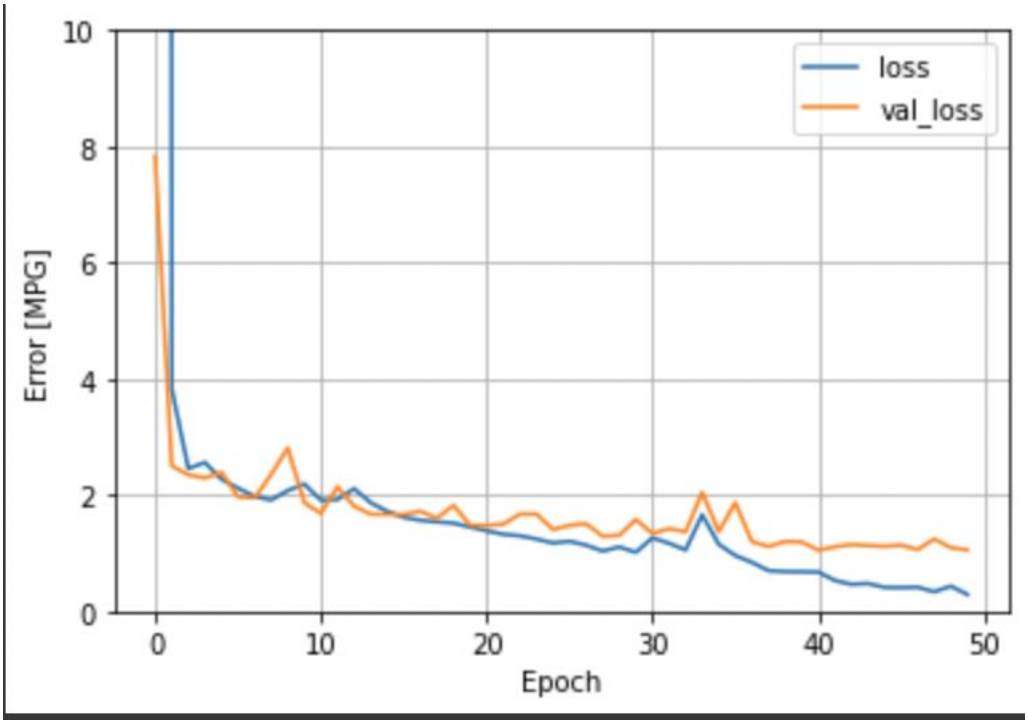


Figure 66: Train/Test Loss Curve. The blue curve represents the training loss, and the orange represents the validation loss.

4.3.2 Baseline Methods

To put our results into perspective, we tested several baselines models to compare with our Convolutional Neural Network. The first baseline we tested was the variance score. The variance score is essentially just predicted the mean altitude for the training dataset. In Figure 67, we can see the variance score is 3.15 which is significantly higher than the MSE scores obtained from the CNN model. The second baseline model was the KNN model using K=1 which produced an MSE of 3.44. Since the CNN received a 3-D input, we flattened the 3-D matrix because the KNN and Random Forest models within the Sci-kit learn library only accept 2-D inputs. The last baseline model we tried was the Random Forest model which had an MSE of 4.18. The CNN model also outperformed the Random Forest model. With these results, it seems the compression and feature learning from the CNN model played a vital role in predicted the altitude change adequately.

Variance	KNN K=1	Random Forest
3.15	3.44	4.18

Figure 67: MSE Scores for Baseline methods to compare with the CNN Model

4.3.3 CNN Model Variations

We tested the CNN model with a different number of CNN layers. We can see in Figure 68 with 1 Convolutional and Pooling layer, the MSE is very high at around 16. When we lowered the number of Convolutional and Pooling layers to 2,3, and 4 we saw a significant decrease in the MSE. The best architecture included six convolutional and pooling layers. Shallow networks weren't enough to learn the features of the high dimensional image matrices especially since they were in RGB format.

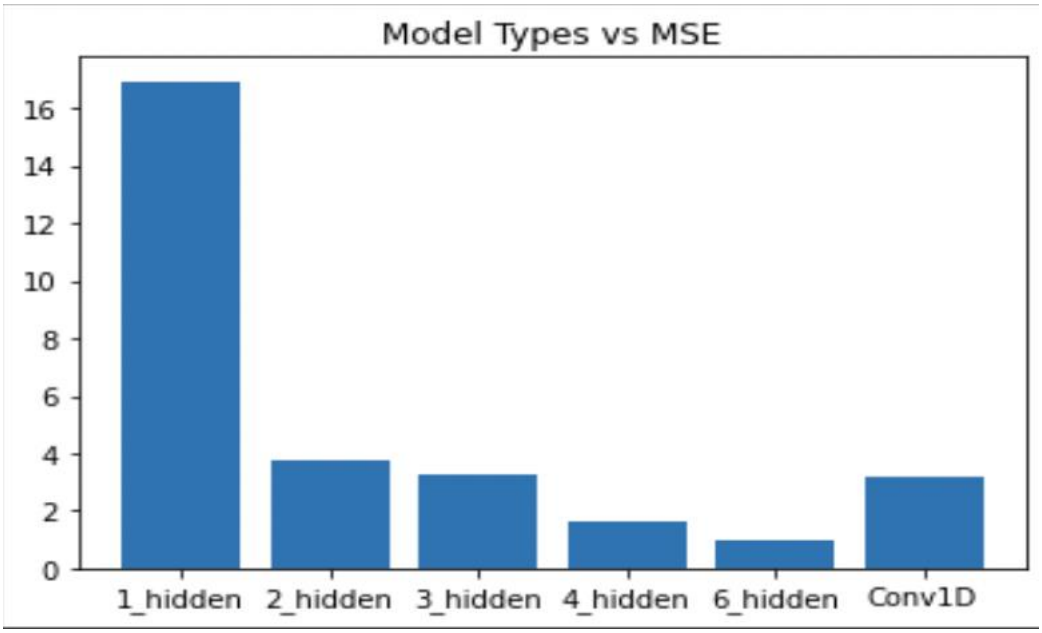


Figure 68: Figure showing the MSE scores for different number of hidden/CNN layers.

We also tested different optimizers and the Adam optimizer had the lowest cross-validated error, so we used that for our final model. The results for each optimizer

can be seen in Figure 69. Adam seemed to perform the best because other optimizers require longer computational times and take longer to converge to a minimum point. In Figure 70, we can see lower learning rates performed better on our dataset. We utilized the 0.001 default Adam learning rate.

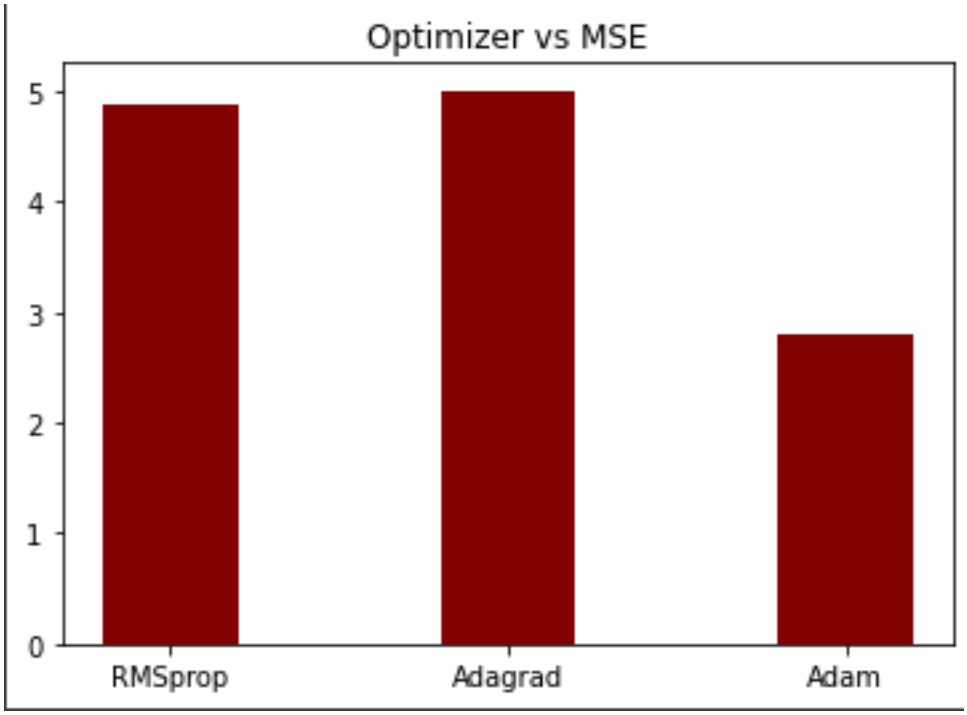


Figure 69: Figure showing the MSE scores for different number of hidden/CNN layers.

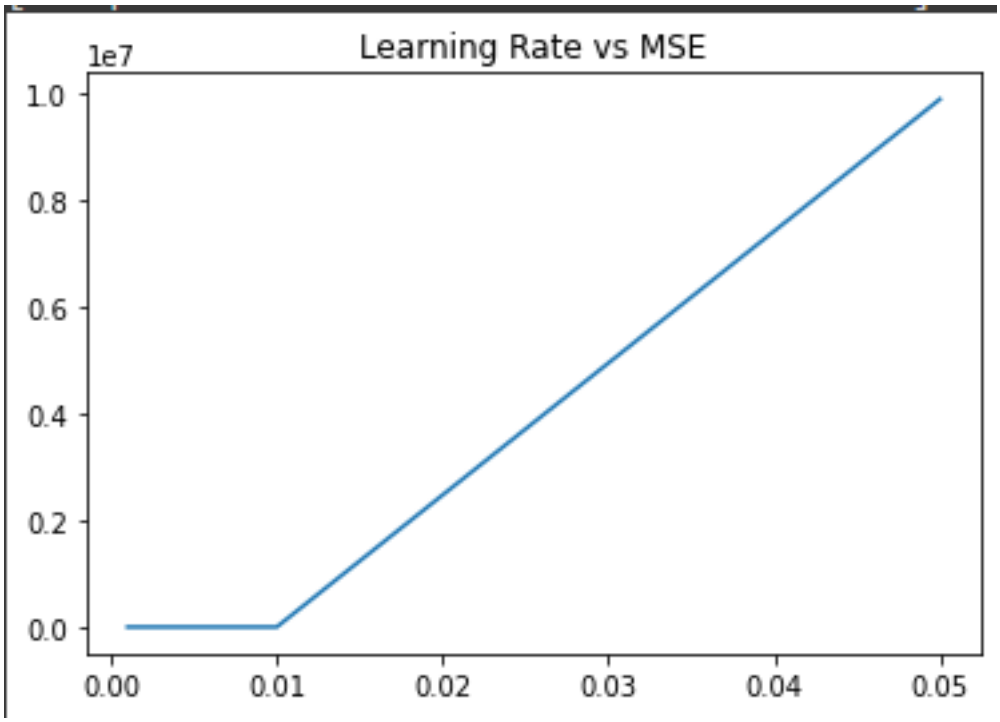


Figure 70: Learning rate vs MSE Graph

5. Future Work

Our work for DEVCOM is a continuation of their partnership with WPI, and we hope to continue this relationship with more MQPs in the future. Since the simulator now has an intuitive user interface and it can recognize objects in the footage, future work could involve giving the simulator additional functionality in different color schemes that could have applications in the field, such as infrared or night vision. This will allow the simulator to recognize objects at night if future teams choose to move in this direction.

Another improvement that could be made by a future team would be to change the simulator so a user can choose their ending location instead of their starting location. This is more helpful for DEVCOM as they know where they want the parafoil to land, and the simulator could help them find the right location to drop from.

Lastly, our neural network could potentially be improved with the use of a varied training set. As it stands, our neural network is trained with two consecutive images taken from a video feed, from which our model can predict the change in altitude over the course of a parafoil drop. However, it may be possible to improve the versatility of the model by adding in training image pairs that are random or not sequential in a video feed. The actual outcome of this and whether it could serve DEVCOM's goals is undecided and could give future teams an avenue to improve our existing project.

6. Conclusion

Our main goal for this project is two-fold. Firstly, we wanted to improve the air drop simulation from last year, properly wrapping it into one neat application that could be easily used by anyone without the need for intimate knowledge of its inner workings. Second, we wanted to continue last year's research into a potential use of utilizing neural networks to supplant the need of GPS for navigating DEVCOM's parafoils.

Both aspects of our project were developed in tandem to be used together in application. For our flight simulator, we were able to fully harness the ability of the unreal engine. This resulted in multiple simulators, one that simulates parafoil drops, and another that simulates a constant altitude horizontal flyby. Both simulators are fully functional and easy to export with easy-to-use menus and settings, making the development of aerial photos. For our neural network, we were able to fully develop a convolutional neural network that could be trained on the simulator's images to predict changes in altitude, longitude, and latitude based on image data. This CNN was able to take in labeled images from our convolutional neural network as well. The CNN architecture with six convolutional and pooling layers performed the best.

7. Appendices

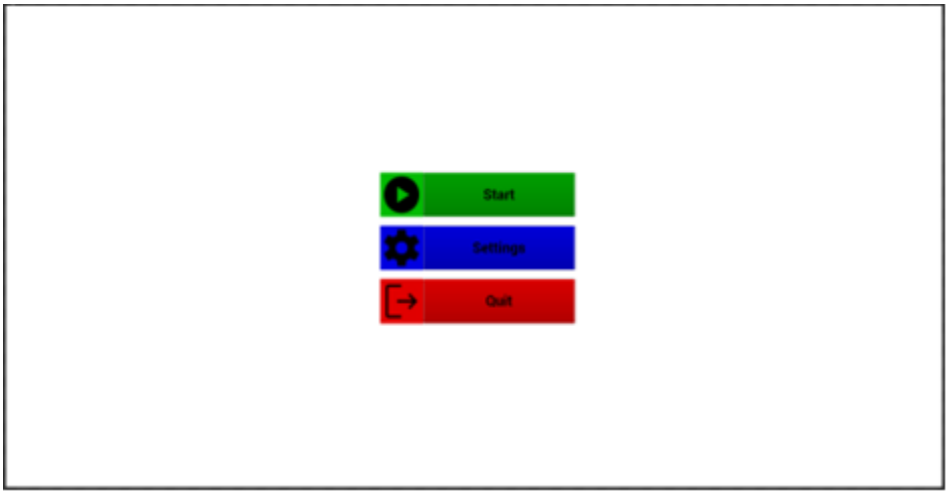
7.1 Using the Simulation Application

7.1.1 Opening the Application

First the simulator .ZIP file must be downloaded on the computer. Once the .ZIP is downloaded, the files must be extracted. This is done by selecting the .ZIP file, then right clicking the file and selecting “Extract all...” from the options. Then select the desired location of the simulator files and click “extract”. Once the files have been extracted there should be a new folder called “WindowsNoEditor”, inside that folder is a file called “Synthetic Parachute Drop Simulator”, by double clicking that file the simulator will launch.

7.1.2 Available Settings

When the application is launched the following screen should be visible.



When the settings button is pressed, the following menu will open where the parameters of the simulation can be set.

The first parameter is the drop-down list of preinstalled flight paths. Clicking "Select Flight Path" will open the drop-down, where the desired flight path can be chosen. Next, the Longitude, Latitude, and Altitude for the desired start location should be typed into the fields indicated below.

The angle at which the simulation can be viewed is set under the "Camera Angle" section. This input is in degrees and will accept inputs from –180 to 180, an input of 0 will result in the camera facing parallel to the horizon while the parachute load is upright. An input of –90 will result in the camera facing directly towards the

ground while the parachute load is upright. The "Number of Screenshots" input allows the user to control the number of screenshots they would like the program to take, this will accept any positive integer. The "Duration in seconds" input allows the user to set the duration of the simulation in seconds, it should be noted that using the screenshot feature may increase the total time it takes to run the program due to stutters. An example of this is if the screenshot causes a 0.5 second delay and the user selects 2 screenshots and a 3 second duration, the actual runtime of the simulation will be 4 seconds. The "Time Zone" input takes an integer from –12 to 12 and represents the desired time zone of the simulation relative to GMT, for example, if the user would like to run the simulation in Worcester, MA, the user should input –5 in the "Time Zone" input. The "Time of Day" input takes multiples of 100 from 0000 to 2400. The input can be interpreted as the first two digits being the desired hour on a 24-hour clock, and the last two digits being the minutes, but the minutes will always be 00. The checkbox right above the "Back" button of the screen controls whether there is fog during the simulated drop.

7.1.3 Features While the Program is Running

While the simulation is running the user has the option to pause the simulation to look around, this is done by pressing "P" on the keyboard while the simulation is running, the simulation can then be resumed by pressing "P" again. While paused, the user can rotate the camera in place by using the mouse, when the simulation is resumed the camera will reset back to the desired camera angle from the menu.

12. Bibliography

[1] C. Dever, L. Hamilton, R. Truax, L. Wholey, K. Bergeron, and G. Noetscher, "Guided-Airdrop Vision-Based Navigation," in *24th AIAA Aerodynamic Decelerator Systems Technology Conference*. [Online]. Available: https://arc.aiaa.org/doi/abs/10.2514/6.2017-3723. [Accessed: 10-Sep-2022].

[2] Q. Yang, Y. Zhang, W. Dai, and S. J. Pan, *Transfer learning*. Cambridge, MA: Cambridge University Press, 2020.

[3] S. Chakraverty, D. M. Sahoo, and N. R. Mahato, "McCulloch–Pitts Neural Network model," *Concepts of Soft Computing*, pp. 167–173, May 2019.

[4] Satellite Navigation - GPS - How It Works — Federal Aviation Administration. [Online]. Available: https://www.faa.gov/about/office_org/headquarters_offices/ato/service_units/techops/navservices/gnss/gps/howitworks [Accessed: 10-Sep-2022].

[5] Z. Fitzgibbon, R. Veetekat, and K. Fabrizio, "Feature recognition from aerial imaging," Available: https://digital.wpi.edu/concern/student_works/gt54kr13r?locale=en. [Accessed: 30-Aug-2022].

[6] J. Scheufele, G. Malabanti, and J. Spitaels, "Simulations and machine learning for parachute navigation." [Online]. Available:

https://digital.wpi.edu/concern/student_works/sj1395264?locale=en. [Accessed: 30-Aug-2022].

[7] "Federal Aviation Administration," 2007. [Online]. Available: https://www.faa.gov/regulations_policies/handbooks_manuals/aviation/media/powered_parachute_handbook.pdf. [Accessed: 03-Nov-2022].

[8] R. Benney, J. Barber, J. McGrath, J. McHugh, G. Noetscher, and S. Tavan, "The new military applications of precision airdrop systems," *Aerospace Research Central*, 2012. [Online]. Available: https://apps.dtic.mil/sti/pdfs/ADA600266.pdf. [Accessed: 27-Jan-2023].

[9] *City Skyscrapers Night Light Road*. (n.d.). Wallpapertip. Retrieved from https://www.wallpapertip.com/wpic/bRhhoh_city-skyscrapers-night-light-road-62141/.

[10] C. Chen, Q. Yan, M. Li, and J. Tong, "Classification of blurred flowers using convolutional neural networks," *Proceedings of the 2019 3rd International Conference on Deep Learning Technologies*, 2019.

[11] K.-M. Koo and E.-Y. Cha, "Image recognition performance enhancements using image normalization," *Human-centric Computing and Information Sciences*, vol. 7, no. 1, 2017.

[12] W. Di, A. Bhardwaj, and J. Wei, *Deep Learning Essentials: Your hands-on guide to the fundamentals of Deep Learning and neural network modeling*. Birmingham, United Kingdom: Packt Publishing, 2018.

[13] J. Brownlee, *Deep Learning for Computer Vision*. Melbourne, Australia: Machine Learning Mastery, 2019.

[14] K. K. Pal and S. K. S., "Preprocessing for Image Classification by Convolutional Neural Networks," *2016 IEEE International Conference on Recent Trends in Electronics, Information & Communication Technology (RTEICT)*, 2016.

[15] S. Brahmbhatt, *Practical opencv: Hands-on Project for Computer Vision on the windows, linux and Raspberry Pi platforms*. Berkeley, CA: Apress, 2013.

[16] F. Visin, K. Kastner, K. Cho, M. Matteucci, A. Courville, and Y. Bengio, "Renet: A recurrent neural network based alternative to Convolutional Networks," *arXiv.org*, 23-Jul-2015. [Online]. Available: https://arxiv.org/abs/1505.00393. [Accessed: 10-Dec-2022].

[17] "Contour detection using opencv (python/C++)," *LearnOpenCV*, 21-Dec-2022. [Online]. Available: https://learnopencv.com/contour-detection-using-opencv-python-c/. [Accessed: 10-Feb-2023].

[18] G. Amer and A. Abushaala, "Edge detection methods | IEEE conference publication | IEEE Xplore," *IEEE Xplore*, 20-Aug-2015. [Online]. Available: https://ieeexplore.ieee.org/document/7210349/. [Accessed: 01-Mar-2023].

[19] J. Prewitt, "Object Enhancement and Extraction," *Picture Processing and Psychopictorics*, pp. 75–149, 1970.

[20] "What is edge detection - an introduction," *Great Learning Blog*, 13-Dec-2022. [Online]. Available: https://www.mygreatlearning.com/blog/introduction-to-edge-detection/. [Accessed: 05-Mar-2023].

[21] G. T. Shrivakshan and C. Chandrasekar, "Comparison of various edge detection techniques used in image processing," *International Journal of Computer Science Issues*, Sep-2012. [Online]. Available: https://www.ijcsi.org/papers/IJCSI-9-5-1-269-276.pdf. [Accessed: 12-Jan-2023].

[22] Liang, J. (n.d.). *CANNY EDGE DETECTION*. Canny Edge Detector. Retrieved March 22, 2023, from https://justin-liang.com/tutorials/canny/

[23] Gonzalez, R. C., & Woods, R. E. (1992). Digital Image Processing. Addison-Wesley. Chapter 3.2.2.2 Laplacian of Gaussian (LoG) Edge Detector.

[24] S. L. Bangare, A. Dubal, P. S. Bangare, and S. T. Patil, "Reviewing otsu's method for image thresholding," *International Journal of Applied Engineering Research*, vol. 10, no. 9, pp. 21777–21783, 2015.

[25] D. V. Widder, *Laplace transform (PMS-6)*. 1941, NJ: Princeton University Press, n.d..

[26] "Apply a Gauss filter to an image with python," *GeeksforGeeks*, 26-Dec-2020. [Online]. Available: https://www.geeksforgeeks.org/apply-a-gauss-filter-to-an-image-with-python/. [Accessed: 07-Mar-2023].

[27] R. Fisher, S. Perkins, A. Walker, and E. Wolfart, "Gaussian Smoothing," *Image Processing Learning Resources*, 2003. [Online]. Available: https://homepages.inf.ed.ac.uk/rbf/HIPR2/gsmooth.htm. [Accessed: 17-Jan-2023].

[28] Kozyrkov, C. (2022, August 19). *Classification, regression, and prediction - what's the difference?* Medium. Retrieved March 22, 2023, from https://towardsdatascience.com/classification-regression-and-prediction-whats-the-difference-5423d9efe4ec

[29] D. Zhu, S. Lu, M. Wang, J. Lin, and Z. Wang, "Efficient precision-adjustable architecture for Softmax function in deep learning," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 67, no. 12, pp. 3382–3386, Dec. 2020.

[30] "Digital Image Interpolation," *Understanding Digital Image Interpolation*, 2020. [Online]. Available: https://www.cambridgeincolour.com/tutorials/image-interpolation.htm. [Accessed: 01-Mar-2023].

[31] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *arXiv.org*, 10-Dec-2015. [Online]. Available: https://arxiv.org/pdf/1512.03385.pdf. [Accessed: 08-Feb-2023].

[32] M. Thoma, "A survey of semantic segmentation," *arXiv.org*, 11-May-2016. [Online]. Available: https://arxiv.org/abs/1602.06541. [Accessed: 01-Mar-2023].

[33] S. Hao, Y. Zhou, and Y. Guo, "A brief survey on semantic segmentation with Deep Learning," *Neurocomputing*, 13-Apr-2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0925231220305476. [Accessed: 01-Mar-2023].

[34] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv.org*, 10-Apr-2015. [Online]. Available: https://arxiv.org/abs/1409.1556. [Accessed: 10-Mar-2023].

[35] Cansiz, S. (2020, November 25). *The math behind training neural networks*. Medium. Retrieved March 22, 2023, from https://towardsdatascience.com/adventure-of-the-neurons-theory-behind-the-neural-networks-5d19c594ca16

[36] J. D. Kelleher, *Deep learning*. Cambridge, MA: The MIT Press, 2019.

[37] GeeksforGeeks. "Understanding Activation Functions in Depth." GeeksforGeeks, n.d., https://www.geeksforgeeks.org/understanding-activation-functions-in-depth/.

[38] D. Choi, C. J. Shallue, Z. Nado, J. Lee, C. J. Maddison, and G. E. Dahl, "On empirical comparisons of optimizers for deep learning," Cornell University, Ithaca, 2020.

[39] Jordan, J. (2023, March 5). *Setting the learning rate of your neural network.* Jeremy Jordan. Retrieved March 22, 2023, from https://www.jeremyjordan.me/nn-learning-rate/

[40] M. Sewak, R. Karim, and P. Pujari, *Practical convolutional neural networks: Implement advanced deep learning models using Python*. Birmingham, United Kingdom: Packt Publishing, 2018.

[41] Y. Lecun, L. Bottou, Y. Bengio and P. Haffner, "Gradient-based learning applied to document recognition," in *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278-2324, Nov. 1998, doi: 10.1109/5.726791.

[42] Nair, A. (2021, April 9). *Deep Q-learning simply explained*. Medium. Retrieved March 22, 2023, from https://ai.plainenglish.io/deep-q-learning-simply-explained-450c35ac954b

[43] R. Shanmugamani and R. Shanmugamani, “5: Semantic Segmentation,” in *Deep Learning for Computer Vision: Expert techniques to train advanced neural networks using TensorFlow and keras*, Birmingham, United Kingdom: Packt, 2018, pp. 129–156.

[44] “Hugging face – the AI community building the future.,” *Hugging Face* –, 2022. [Online]. Available: https://huggingface.co/. [Accessed: 15-Feb-2023].

[45] “Tensorflow,” *TensorFlow*. [Online]. Available: https://www.tensorflow.org/. [Accessed: 20-Mar-2023].

[46] M. Harrison, *Machine learning pocket reference: Working with structured data in Python*. Sebastopol, CA: O'Reilly Media, 2019.

[47] Jaccard, Paul. (1901). Etude de la distribution florale dans une portion des Alpes et du Jura. Bulletin de la Societe Vaudoise des Sciences Naturelles. 37. 547-579. 10.5169/seals-266450.

[48] S. N. Kumar, A. L. Fred, A. K. Haridhas, and S. Varghese, "Medical image edge detection using Gauss gradient operator," *ResearchGate*, Jan-2017. [Online]. Available: https://www.researchgate.net/publication/317754223_Medical_image_edge_detection_using_gauss_gradient_operator. [Accessed: 04-Mar-2023].

[49] U. Sinha, "The Sobel and laplacian edge detectors," *AI Shack*, 2010. [Online]. Available: https://aishack.in/tutorials/sobel-laplacian-edge-detectors/. [Accessed: 04-Mar-2023].

[50] "2D & 3D Data Labelling," *Segments.ai*, 2022. [Online]. Available: https://segments.ai/. [Accessed: 01-Mar-2023].

[51] G. James, D. Witten, T. Hastie, and R. Tibshirani, "An introduction to statistical learning," *SpringerLink*, 2013. [Online]. Available: https://link.springer.com/book/10.1007/978-1-4614-7138-7. [Accessed: 16-Jan-2023].

[52] *Segments.ai*. [Online]. Available: https://segments.ai/. [Accessed: 20-Mar-2023].

[53] "Joint Precision Airdrop System (JPADS)," *USAASC*, 2023. [Online]. Available: https://asc.army.mil/web/portfolio-item/cs-css-joint-precision-airdrop-system-jpads/. [Accessed: 02-Feb-2023].

[54] W. Wailes and N. Harrington, "The Guided Parafoil Airborne Delivery System program," *American Institute of Aeronautics and Astronautics, Inc.*, 2012. [Online]. Available: https://faculty.nps.edu/oayakime/ADSC/. [Accessed: 05-Dec-2022].

[55] D. Carter, S. George, P. Hattis, L. Singh, and S. Tavan, "Autonomous Guidance, navigation and control of large parafoils," *Aerospace Research Central*, 11-Nov-2012. [Online]. Available: https://arc.aiaa.org/doi/10.2514/6.2005-1643. [Accessed: 12-Dec-2022].

[56] "Jamming and Spoofing 9 - Maritime Global Security," *Maritime Global Security*, 2019. [Online]. Available: https://www.maritimeglobalsecurity.org/media/1043/2019-jamming-spoofing-of-gnss.pdf. [Accessed: 15-Jan-2023].

[57] "How single-shot detector (SSD) works?," *ArcGIS API for Python*, 2019. [Online]. Available: https://developers.arcgis.com/python/guide/how-ssd-works/. [Accessed: 07-Feb-2023].

[58] Y. Benezeth, P.-M. Jodoin, B. Emile, H. Laurent, and C. Rosenberger, "(PDF) Comparative Study of background subtraction algorithms - researchgate," *ResearchGate*, Jul-2010. [Online]. Available:

https://www.researchgate.net/publication/257365159_Comparative_study_of_background_subtraction_algorithms. [Accessed: 07-Feb-2023].

[59] J. Feng, P. Liu, and Y. K. Kim, "Foreground detection based on Superpixel and semantic segmentation," *Computational intelligence and neuroscience*, 31-Aug-2022. [Online]. Available: https://pubmed.ncbi.nlm.nih.gov/36093472/. [Accessed: 08-Feb-2023].

[60] J. Hsu, "Military tests robo-parachute delivery needing no GPS," *IEEE Spectrum*, 02-Feb-2016. [Online]. Available: https://spectrum.ieee.org/military-tests-roboparachute-delivery-needing-no-gps. [Accessed: 07-Mar-2023].

[61] W. Yao, D. Zhou, L. Zhan, Y. Liu, Y. Cui, S. You, and Y. Liu, "GPS signal loss in the wide area monitoring system: Prevalence, impact, and solution," *Electric Power Systems Research*, 19-Mar-2017. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0378779617300950. [Accessed: 07-Mar-2023].

[62] I. Sobel and G. Feldman, "(PDF) an isotropic 3x3 image gradient operator - researchgate," *ResearchGate*, Aug-2015. [Online]. Available: https://www.researchgate.net/publication/281104656_An_Isotropic_3x3_Image_Gradient_Operator. [Accessed: 15-Feb-2023].

[63] J. Canny, "A computational approach to edge detection," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. PAMI-8, no. 6, pp. 679–698, Nov. 1986.

[64] L. Roberts, "Machine perception of three-dimensional solids," *Semantic Scholar*, 01-Jan-1963. [Online]. Available: https://www.semanticscholar.org/paper/Machine-Perception-of-Three-Dimensional-Solids-Roberts/ab5387cf077f5b97c7dd08845c006e5c1ec89ff5. [Accessed: 07-Mar-2023].

[65] B. Green, "Canny Edge Detection Tutorial," 2002. [Online]. Available: https://web.archive.org/web/20160324173252/http://dasl.mem.drexel.edu/alumni/bGreen/www.pages.drexel.edu/_weg22/can_tut.html. [Accessed: 16-Jan-2023].

[66] R. A. Haddad and A. N. Akansu, "A class of fast gaussian binomial filters for speech and image processing," *IEEE Xplore*, 1991. [Online]. Available: https://ieeexplore.ieee.org/document/80892. [Accessed: 17-Jan-2023].

[67] R. Fisher, S. Perkins, A. Walker, and E. Wolfart, "Sobel Edge Detector," *Image Processing Learning Resources*, 2003. [Online]. Available: https://homepages.inf.ed.ac.uk/rbf/HIPR2/sobel.htm [Accessed: 17-Jan-2023].

[68] "Lerp," *Unreal Engine Documentation*, 2023. [Online]. Available: https://docs.unrealengine.com/4.27/en-US/API/Runtime/Core/Math/FMath/Lerp/. [Accessed: 10-Mar-2023].

[69] "Using a geospatially accurate sun," *Cesium*, 2022. [Online]. Available: https://cesium.com/learn/unreal/unreal-geospatially-accurate-sun/. [Accessed: 10-Oct-2022].

[70] V. Jagannath, *Random Forest Simplified*. 2017.

[71] Information Resources Management Association, Ed., *Research anthology on Artificial Neural Network Applications*. Hershey, PA: Engineering Science Reference an imprint of IGI Global, 2022.