3-2022

Detection and Identification of Cellphone Emitted Light Detection and Ranging Light in Security Camera Video Footage

Tristan V. Creek

DETECTION AND IDENTIFICATION OF CELLPHONE EMITTED LIGHT DETECTION AND RANGING LIGHT IN SECURITY CAMERA VIDEO FOOTAGE

THESIS

Tristan V. Creek, 2d Lieutenant, USAF

AFIT-ENG-MS-22-M-019

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT-ENG-MS-22-M-019

DETECTION AND IDENTIFICATION OF CELLPHONE EMITTED LIGHT DETECTION AND RANGING LIGHT IN SECURITY CAMERA VIDEO FOOTAGE

THESIS

Presented to the Faculty

Department of Electrical and Computer Engineering

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

in Partial Fulfillment of the Requirements for the

Degree of Master of Science in Cyber Operations

Tristan V. Creek, B.S. Computer Science

2d Lieutenant, USAF

March 24, 2022

AFIT-ENG-MS-22-M-019

DETECTION AND IDENTIFICATION OF CELLPHONE EMITTED LIGHT DETECTION AND RANGING LIGHT IN SECURITY CAMERA VIDEO FOOTAGE

THESIS

Tristan V. Creek, B.S. Computer Science
2d Lieutenant, USAF

Committee Membership:

Barry E. Mullins, Ph.D., P.E.
Chair

Timothy H. Lacey, Ph.D., CISSP
Member

Frankie A. Cruz, M.S., Capt, USAF
Member

Abstract

The prevalence of Light Detection and Ranging (LiDAR) sensors in consumer cellphones has ushered in an era of access to technologies previously inaccessible to the average person. Many people may walk around with a cellphone in their pocket every day unaware of its ability to capture 3D object scans, accurately measure distances in 3D space, and even provide limited range night vision. These abilities provide the common person with access to an advanced data collection toolkit which they may leverage for nefarious activities. The ability to detect LiDAR activity from cellphones can benefit sensitive environments that want to detect when people capture 3D scans or use other LiDAR capabilities. Although LiDAR sensors emit light in the infrared spectrum invisible to the naked eye, many security cameras possess the ability to capture infrared light reflected off surfaces to detect LiDAR light in an environment. Once detected and recorded to video footage, current classification methods fail to identify the LiDAR light. Therefore, this research develops a methodology to detect LiDAR light in security camera video footage and identify it as cellphone LiDAR. A proposed image processing technique separates the LiDAR light pattern from the background of a video frame. The extracted LiDAR light pattern is then permuted many ways to simulate how it may appear in video footage. A neural network trains to identify these permutations which results in a neural network that classifies whether or not a given image contains the LiDAR light pattern. In a dark environment with a cellphone and security camera positioned 1 meter from a wall, the trained neural network correctly identifies 99.7% of images containing cellphone LiDAR reflected off the wall.

Table of Contents

Page

Page

Page

List of Figures

List of Tables

List of Acronyms

Abbreviation		Page
1D	1-Dimensional	8
2D	2-Dimensional	8
3D	3-Dimensional	1
4D	4-Dimensional	103
ARP	Address Resolution Protocol	52
bps	bits per second	12
CCD	Charged Coupled Device	10
CMOS	Complementary-Metal-Oxide Semiconductor	10
COTS	Commercial-Off-The-Shelf	1
DARPA	Defense Advanced Research Projects Agency	40
dB	decibels	13
DFT	Discrete Fourier Transform	27
DOE	Diffractive Optical Element	9
FN	False Negative	124
FP	False Positive	124
FPS	Frames Per Second	11
IDE	Integrated Development Environment	35
IP	Internet Protocol	52
KNN	K-Nearest Neighbors	27
LED	Light Emitting Diode	5
LiDAR	Light Detection and Ranging	1
MOG	Mixture of Gaussians	27
NIR	Near-Infrared	9

Abbreviation		Page
nm	nanometer	9
ns	nanosecond	37
PoE	Power over Ethernet	52
ReLU	Rectified Linear Unit	29
s	seconds	11
SE	Structuring Element	16
SGD	Stochastic Gradient Decent	31
SPI-3D	Standoff Precision Identification in Three Dimensions	40
SUT	System Under Test	122
SWOM	Sliding Window Otsu's Method	48
TN	True Negative	124
ToF	Time of Flight	9
TP	True Positive	124
URL	Uniform Resource Locator	64

DETECTION AND IDENTIFICATION OF CELLPHONE EMITTED LIGHT DETECTION AND RANGING LIGHT IN SECURITY CAMERA VIDEO FOOTAGE

I. Introduction

1.1 Overview

Although Light Detection and Ranging (LiDAR) may not be a household name, it has been used in many different technologies for decades [1]. The near-infrared light used in LiDAR possesses characteristics like material penetration that enable many different applications such as range detection, forest inventory, urban development, and 3-Dimensional (3D) scanning [2][3]. Although LiDAR supports many academic and industrial research pursuits, it has also supported many military projects for decades which indicates its potential adversarial use [4][5]. This posed less of a threat 5 years ago when LiDAR required costly specialized hardware, but recent advances in LiDAR hardware has enabled production of Commercial-Off-The-Shelf (COTS) cellphones with LiDAR sensors [1]. Although these LiDAR sensors have limited range and functionality compared to high power LiDAR sensors such as those mounted to airplanes for topography mapping, cellphone LiDAR sensors have ushered in an era of access to new technologies for consumers worldwide. Many people may walk around with a cellphone in their pocket every day unaware of its ability to 3D scan objects, accurately measure distances in 3D space, and even provide limited range night vision [6][7][8].

These functionalities seem benign at face value, but they provide potential in-

telligence collection avenues for adversaries. 3D scanning a building or recording night vision video in an unlit area may be considered a suspicious or nefarious activity by those in charge of protecting certain sensitive environments. It therefore behooves these protectors to detect and identify such LiDAR activity. Specialized cameras, such as infrared cameras, can easily identify the near-infrared light that LiDAR sensors emit, but this requires niche equipment which may not be applicable for general purpose surveillance [9]. Environments where LiDAR activity may be considered suspicious likely already use general purpose security cameras to monitor other suspicious activity, so the ability to utilize these security cameras to detect and identify LiDAR activity reduces the cost and increase the ease of implementation. The ability to detect LiDAR relies on the security camera's ability to capture LiDAR light while the ability to identify LiDAR depends on the method used to process the captured security camera footage. This research develops and evaluates a method to capture LiDAR light in security camera footage and identify whether or not it contains cellphone LiDAR light.

1.2 Problem Statement

LiDAR light may project upon a variety of environments with different lighting, textures, and shapes. The ability to detect and identify LiDAR light in a wide range of environments builds upon the ability to detect and identify LiDAR light in a simple environment. The detection of LiDAR light relies on the interaction between the security camera and the environment, so the most effective method for capturing LiDAR light with a security camera in a simple environment must first be identified. Once captured and recorded to video footage, the identification of LiDAR light depends on the ability to recognize characteristics of cellphone LiDAR light as captured in video footage. Therefore, the video processing methodology must

recognize the key characteristics of cellphone LiDAR light to identify its presence in a video.

1.3 Research Goals

This research aims to develop a methodology that accurately classifies cellphone LiDAR light when captured in security camera footage. To achieve this goal, three questions must be addressed.

- Can a security camera reliably capture cellphone LiDAR light?
- Is there a reliable method to identify captured cellphone LiDAR light in a video?
- What factors most heavily influence the identification of cellphone LiDAR light in a video?

1.4 Research Hypothesis

This research proposes two hypotheses to test the research goals:

1. In an environment with controlled textures and light levels, a security camera configuration exists that captures cellphone LiDAR light projected into the environment the majority of the time.
2. If video footage of cellphone LiDAR light is captured in an environment with controlled textures and light levels, each frame of the video can be processed to identify whether or not it contains cellphone LiDAR light with an accuracy above 50%.

The classification of cellphone LiDAR light in a video frame is a binary choice: LiDAR light either is or is not present. Therefore, a binary classification methodology

must classify LiDAR light with an accuracy above 50% to be considered effective in any regard.

1.5 Approach

This research experimentally identifies security camera settings that reliably capture cellphone LiDAR light through trial and error. This process is conveniently more trial than error because the design choices require certain settings that result in the reliable capture of LiDAR light. This also reveals the geometrical pattern of the cellphone LiDAR light used to identify it in images.

With the camera settings identified, the developed method processes the image frames from a video and identifies if they contain cellphone LiDAR light. This method consists of processing recorded images to emphasize the geometrical pattern of cellphone LiDAR light. A neural network trains to identify simulated projections of the cellphone LiDAR light pattern which emulates how the LiDAR light may appear in video footage when pointed in different directions. The trained neural network predicts whether or not a processed image contains cellphone LiDAR light.

1.6 Assumptions and Limitations

1.6.1 Light Level

The security camera must enable night mode to capture LiDAR light in this research. The system design therefore assumes that the security camera records footage in a low light environment that requires the camera to enable night mode in order to effectively monitor the visible light in the environment. This allows the security camera to observe the visible environment for normal security monitoring purposes and capture LiDAR light. To enforce this assumption, the system simulates a low light environment in a room absent of natural and unnatural light except for a

1000 Lumen Light Emitting Diode (LED) light bulb placed approximately 3 meters from the test surface upon which a cellphone projects LiDAR light. This results in a maximum surface illumination of 35 Lux.

1.6.2 Surfaces

External environments captured by security cameras may contain a variety of surfaces such as brick, concrete, metal, glass, and more. Brick, concrete, and rough metals fall into the category of Lambertian surfaces due to their rough texture while visibly reflective materials such as glass and smooth metals meet the criteria of Specular surfaces [10].

Specular ("smooth") surfaces reflect light in a single direction dependent on the incident angle at which the source light casts upon the surface. Lambertian ("rough") surfaces diffuse single points of light in various directions within a 3D sphere-like area with non-uniform intensities. This lack of uniformity results from the partially specular behavior of Lambertian surfaces that produces a specular lobe and specular spike seen in Figure 1. As a Lambertian surface trends towards smoothness, the specular spike dominates the reflectance until it reflects the exact same as a Specular surface. Conversely, the specular spike disappears as a Lambertian surface trends towards roughness [10].

The principles of LiDAR require that light emitted from the LiDAR source reflects off the target object and returns to the LiDAR measurement device typically placed adjacent to the emission source [2]. Unless the light emission source points perpendicular to a Specular surface, said surface does not reflect the light back to the LiDAR measurement device. To ensure that all surfaces diffuse LiDAR light, the research environment only contains Lambertian surfaces and excludes Specular surfaces such as windows and mirrors that may distort the LiDAR pattern.

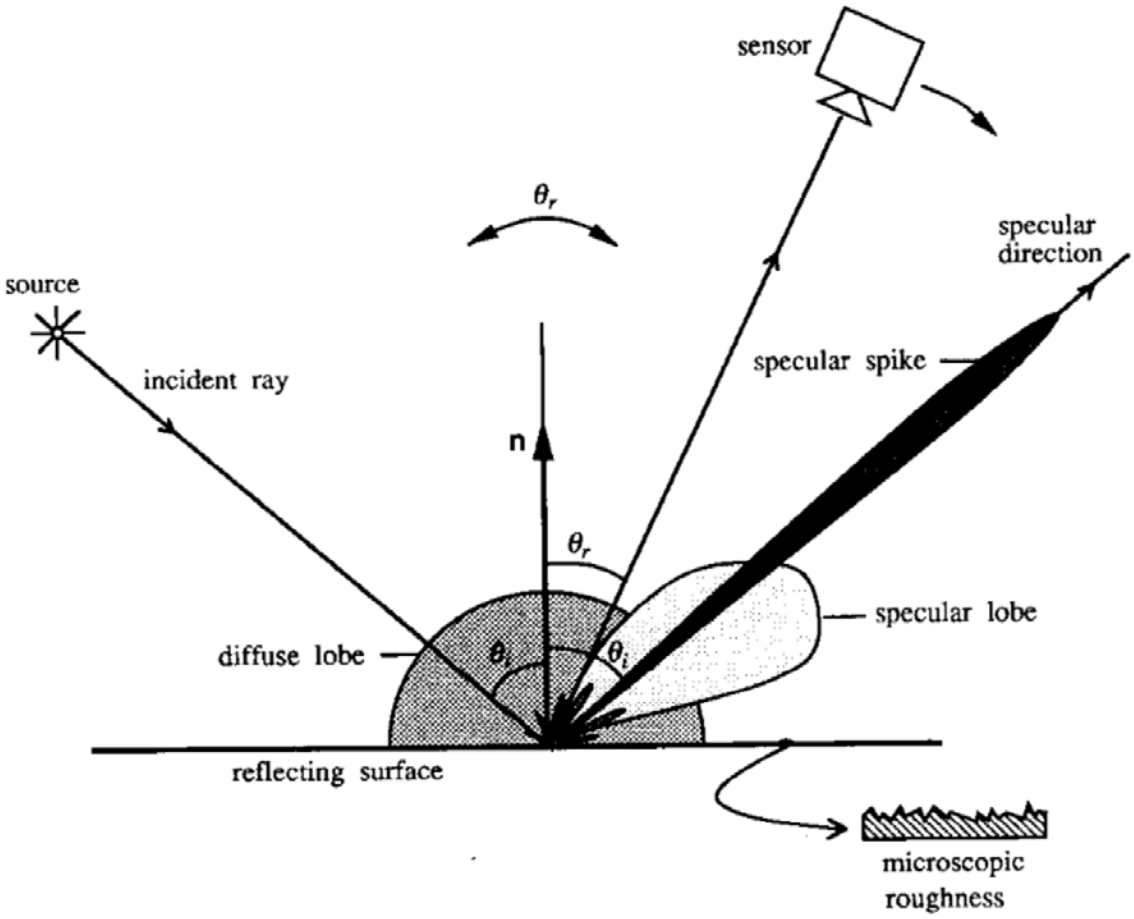

Figure 1: Lambertian surface reflection. [10]

1.7 Research Contributions

This research develops a framework to capture cellphone LiDAR light with a security camera, record it to a video, process the video frames to emphasize the cellphone LiDAR light pattern, and predict the presence of cellphone LiDAR light in these processed video frames with a neural network trained on a dataset generated from a single image of the cellphone LiDAR light pattern. The neural network comes from a pattern classification paper [11], but the creation of training data for the neural network as well as the image processing methodology that emphasizes the LiDAR pattern are both novel.

1.8 Thesis Overview

This thesis contains six chapters. Chapter II defines and elaborates upon the technologies, tools, and related research concepts necessary to understand and complete this research. Chapter III describes the design of the system that accomplishes this research as well as the tools and devices that support the design. Chapter IV presents the experimental design and statistical analysis used to evaluate the system. Chapter V analyzes the results of the experiments and statistical analysis. Chapter VI concludes this research with its results and limitations as well as proposals for future work.

II. Background and Related Research

2.1 Overview

This chapter first explains the fundamental principles of LiDAR in Section 2.2. The affect of these principles on the security camera settings is explained in Section 2.3 along with other notable camera characteristics. Section 2.4 elaborates on how various image processing techniques transform images, and Section 2.5 provides background information on neural networks and how they can be used to classify images. Section 2.6 provides background information on the tools that support this research followed by a discussion of related research in Section 2.7 and a concluding summary of this chapter in Section 2.8.

2.2 LiDAR Principles

LiDAR sensors are comprised of three primary components: the laser source, the receiver, and the optical system for pointing the LiDAR [1]. This research utilizes a security camera as the receiver and therefore disregards the characteristics of the LiDAR receiver. The following sections discuss the dimensionality produced by the optical pointing system, the Time of Flight principles of the laser source, and the light pulsing characteristics of the laser source.

2.2.1 3D LiDAR

A 1-Dimensional (1D) LiDAR sensor emits a single beam of light that finds the distance to a single point of reflection. A 2-Dimensional (2D) LiDAR sensor emits light across a single plane that finds the distance to each point where the plane intersects with the 3D environment. A 3D LiDAR sensor emits light across two orthogonal planes which allows it to find the distance to any location in 3D space.

Although the laser source often plays a role in dimensionality, it is the optical pointing element that controls the dimensionality overall. Optical pointing elements steer light from the laser source to produce 3D scanning behavior. LiDAR sensors on small, mobile devices use solid state optical pointing elements with no moving parts to decrease size and increase robustness in Time of Flight LiDAR sensors [12].

2.2.2 Time of Flight

Time of Flight (ToF) LiDAR sensors measure the distance to an object based on the reflection of Near-Infrared (NIR) light. NIR light is light with a wavelength in the 800-1100 nanometer (nm) range [13]. The reflected NIR light allows the LiDAR source to calculate the distance directly or indirectly.

Direct ToF LiDAR sensors rely on the known speed of light to measure the distance from the laser source to an object. The laser source may consist of an array of individual light emitting diodes that shine through a Diffractive Optical Element (DOE) which multiplies the light points to cover a larger field of view [14]. The LiDAR source measures how long it takes the NIR light to reflect off a surface back to the receiver and calculates the distance [3][15].

Rather than directing the array of diodes through a DOE, indirect ToF LiDAR sensors direct the NIR light through a diffusing element to flood the environment with a blanket of NIR light instead of individual NIR light points. It then compares the phase of the reflected light to the phase of the emitted light to calculate the distance to the object [16].

Although direct and indirect ToF LiDAR sensors illuminate the environment differently, they both illuminate the environment with high frequency pulsing that requires advanced sensors to adequately capture.

2.2.3 Light Pulsing

Some ToF LiDAR sensors emit a continuous stream of NIR light, but portable hardware such as cellphones often implement energy-efficient methods of pulsing the NIR light nanoseconds at a time thousands of times per second [1][3]. In direct ToF LiDAR sensors, a diode activates a timer upon illumination to record the round-trip time of these high speed NIR light pulses [16]. In indirect ToF LiDAR sensors, a time gated Complementary-Metal-Oxide Semiconductor (CMOS) or Charged Coupled Device (CCD) camera captures the NIR light for use in calculation of the phase shift. These direct ToF diodes and indirect ToF CMOS/CCD cameras are designed specifically for high-performance applications such as LiDAR sensors. They can capture the granular characteristics of the NIR light (pulse frequency, pulse width, etc.) that CMOS sensors found in common hardware like security cameras fail to capture [17].

2.3 Camera Operating Principles

CMOS image sensors convert light to electronic signals [18]. Although they have been around for many years, many organizations continue to improve them and build them into COTS hardware such as digital cameras, medical equipment, and security cameras [19]. Most of these COTS CMOS security cameras capture light beyond the end of the visible light spectrum at 780 nm which allows them to capture NIR light from LiDAR sensors and then record it to a video within the visible spectrum [13][20]. Although CMOS security cameras can technically record captured NIR light to a color or grayscale video, most security cameras possess a physical or software NIR light filter to prevent most (if not all) NIR light from distorting videos [13]. However, enabling night mode on a security camera removes the NIR filter to allow more light to reach the sensor for better visibility in low light environments [21]. Therefore, most COTS CMOS security cameras best capture NIR light in night mode which

they record in grayscale. This capture and recording of NIR light in grayscale is affected further by the shutter speed and frame rate of the security camera.

2.3.1 Shutter Speed

The shutter speed is the amount of time a camera's shutter remains open which controls the amount of light it captures for a given frame. A camera with a shutter speed of 1/20 seconds (s) captures 1/20 of a second of light for each frame [22]. These seemingly short shutter speeds capture light for much longer than the length of a single LiDAR light pulse which only lasts a few nanoseconds [4]. Each light pulse is multiple orders of magnitude shorter than the shortest shutter speed offered on the target camera of this research which is 1/10,000 s (100,000 nanoseconds) [23]. Therefore, even the shortest shutter speed provides a high probability of capturing thousands of LiDAR light pulses each frame which often looks like a solid light in video footage, similar to an illuminated street lamp or light bulb [24]. Unfortunately, this also means that many COTS CMOS cameras fail to capture all of the unique pulse characteristics of LiDAR light which may differentiate it from other light sources in video footage.

2.3.2 Frame Rate

The frame rate, or Frames Per Second (FPS), is the frequency at which a camera records frames during a time frame of one second. A camera recording at 30 FPS records 30 individual frames per second, and each frame captures light for the length of the shutter speed [22]. Therefore, the total length of light captured can be described by

$$T = FPS * SHUT \tag{1}$$

where T is the total length of light captured in one second, FPS is the frame rate, and SHUT is the shutter speed. Since T is the length of light captured in one second, the product of FPS and SHUT cannot exceed one second. The average security camera records at 15 FPS which limits the longest shutter speed to 1/15 s [25]. Section 2.3.1 establishes the shortest shutter speed of 1/10,0000 s, so the range of acceptable shutter speeds is 1/10,000 s to 1/15 s. However, cameras need to use a longer shutter speed at night to capture more light for a clearer picture [26], so this research limits the shutter speed to the longest possible shutter speed of 1/15 s.

2.3.3 Bit Rate

The bit rate of a camera describes the rate, measured in bits per second (bps), at which a camera can process captured light to footage and export it to an external storage location [27]. The bit rate affects the quality of recorded video footage, so the maximum bit rate that does not exceed storage limits is preferred [28].

2.3.4 Rolling Shutter

Many CMOS cameras utilize a rolling shutter to capture images rather than a global shutter. While a global shutter captures light at every pixel at the same time, a rolling shutter captures rows of pixels, typically starting at the top of the image, and progresses down the image until it captures the entire image. The environment may change during the capture of a single image causing inconsistency among the rows [26]. This commonly distorts propellers in video footage due to their high rate of rotation [29]. In a similar vain, cameras may capture high frequency light pulses such as LiDAR light in unexpected patterns depending on how the shutter speed syncs with the LiDAR pulse frequency.

2.3.5 Gain

Gain, measured in decibels (dB), amplifies the signal of the captured light before recording it to an image. This improves the visibility of objects in dark settings, but it also improves the visibility of noise. Although some cameras may capture images with a single gain value, other cameras implement an adaptive gain which, given a range of potential gain values, attempts to select a gain value that minimizes the noise in an image [30].

2.3.6 Lux

Although not a direct characteristic of a camera, the footage a camera captures is directly influenced by the light level of the environment. In an environment illuminated by artificial light, the light level measured in Lumens describes the current of the light source. This describes the total light output of a light source, but Lumens recorded across a specific area measures Lux. One Lux equals one Lumen per square meter [31].

2.3.7 Captured LiDAR Characteristics

As detailed in Section 2.2, the LiDAR light contains key characteristics that differentiate it from a light source in the visible spectrum like a light bulb. These characteristics are wavelength, pulse frequency, pulse length, and shape. Section 2.3 establishes that CMOS security cameras record NIR light in grayscale which makes it indifferentiable from a common light source based only on grayscale color [13]. Furthermore, Section 2.3.1 establishes that cameras do not possess short enough shutter speeds to capture the pulse speed or pulse length characteristics of LiDAR light. The only characteristic left to differentiate the LiDAR light by is its shape. Shape refers to the individual light points projected from the DOE in a direct ToF LiDAR sensor and

the blanket of light projected from the diffusion element of an indirect ToF LiDAR sensor.

Note the initial state of the wall as depicted in Figure 2 in absence of direct or indirect LiDAR light. As explained in Section 2.2.2, direct ToF LiDAR from an iPhone 12 Pro Max projects an array of light points onto the wall in Figure 3 whereas indirect ToF LiDAR from a Galaxy S20 Ultra 5G floods light across the wall in Figure 4. The direct ToF LiDAR light creates a unique pattern recognizable at a glance while the indirect ToF LiDAR light looks similar to a visible light flashlight. Although the indirect ToF LiDAR projects light in a rounded square shape, the shape lacks the adequate uniqueness seen in the direct ToF LiDAR light shape for use in distinguishing it from a common light source.

Figure 2: Wall absent of LiDAR light

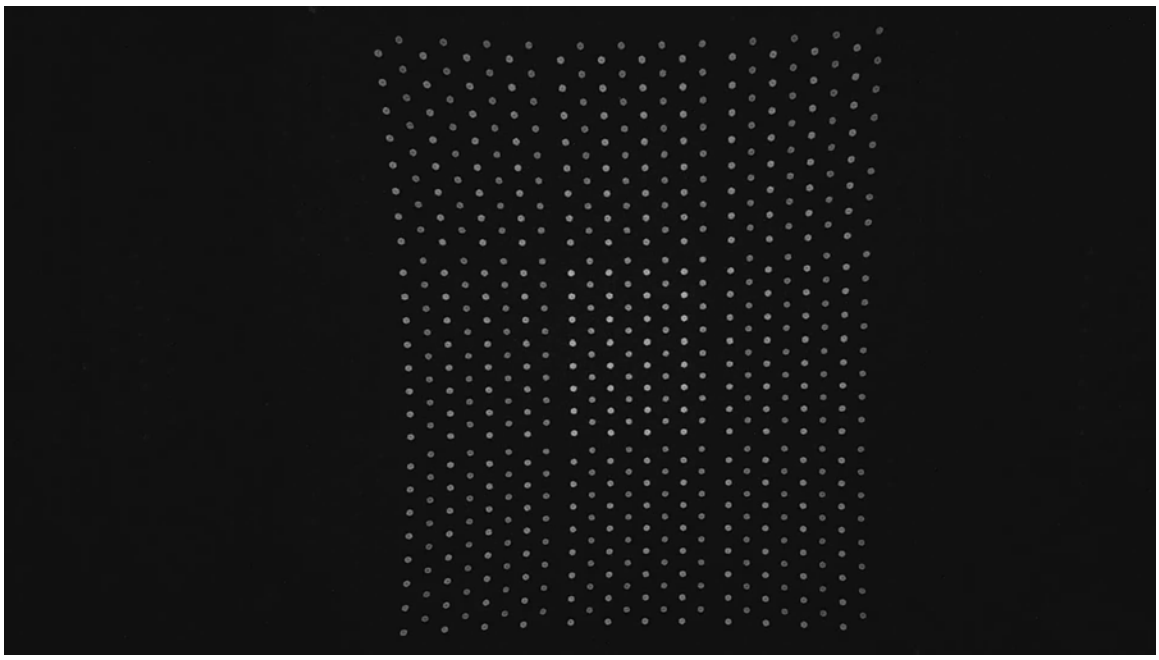

Figure 3: Direct ToF LiDAR projected on wall

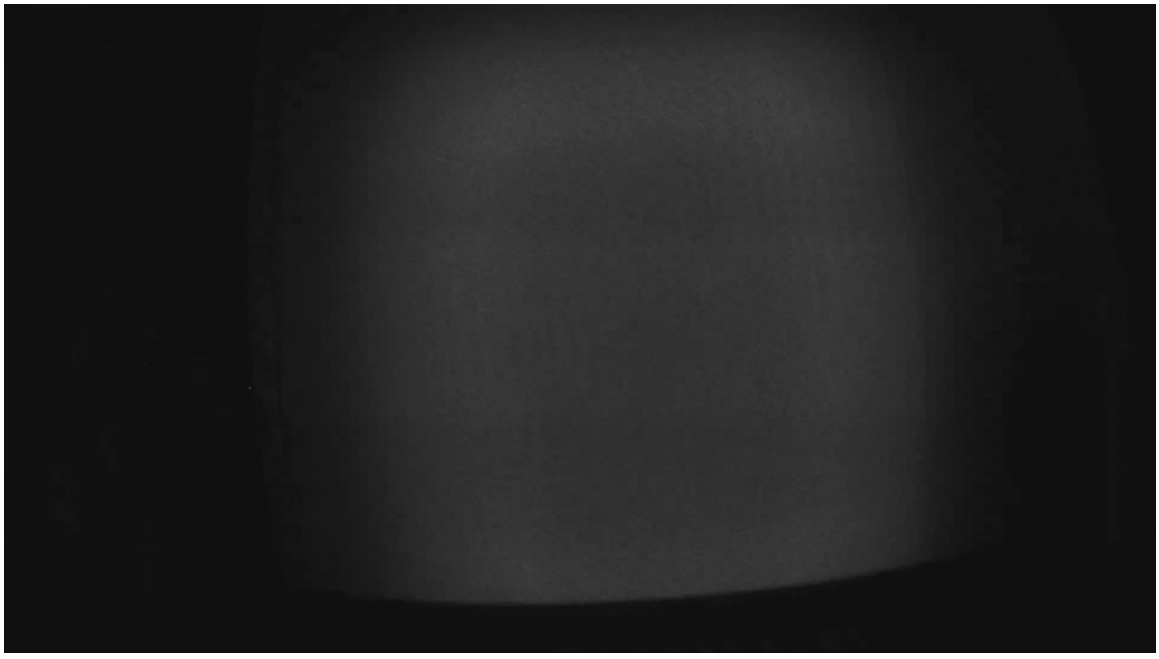

Figure 4: Indirect ToF LiDAR projected on wall

2.4 Image Processing Techniques

Digital image processing refers to the use of a digital computer to process a digital image [32]. A digital image is the 2D function f(x, y) which returns a color value when provided two spatial coordinates x and y. These coordinates locate a single pixel. A pixel's color value may be represented in various formats in a color image, but grayscale images store a single, finite, discrete value for each pixel, sometimes called the pixel intensity. Changing pixel intensities based on certain criteria facilitates many advanced applications such as noise removal, object isolation, small object targeting, and more [33]. Many of these applications rely on morphological operations to define the criteria upon which pixel intensities change.

2.4.1 Morphological Operations

Mathematical morphology operations define how to change the values of pixels in a digital image. The definitions are written in set theory where sets represent objects in an image. A grayscale image must be represented by sets in a 3D integer space with two dimensions for the location of each pixel and one dimension for the intensity. Pixels in binary images contain either the value 0 or 1, so they can instead be represented by sets in a 2D integer space where each element contains the coordinates of the pixels with the value 1. These pixels are typically referred to as the foreground pixels whereas the pixels with the value 0 are the background pixels [32].

Morphological operations use two types of sets: object sets and Structuring Element (SE) sets. In binary images, the object sets only contain foreground pixels whereas SE sets can contain foreground and background pixels. A SE must be a rectangle, but morphological operations ignore the background pixels which allows for control of the effective shape of the SE [32].

Intuitively, morphological operations treat the SE as a window that slides across

the image and operates on the pixel values based on the shape of the SE. This general operation requires two important concepts used extensively by many different morphological operations: set reflection and translation. The reflection of a set B is defined as

$$\hat{B} = \{w | w = -b, \text{ for } b \in B\} \tag{2}$$

where $\hat{B}$ is the set of points w that are the opposites of the points b from set B. If B is a set of 2D points in the format (x, y), then $\hat{B}$ is the set of reflected 2D points $(-x, -y)$. This rotates an object 180 degrees around its origin. In the case of a SE, this rotates both the foreground and the background pixels [32].

Given a point $z = (z_1, z_2)$, the translation of a 2D set B by z is defined as

$$(B)_z = \{c | c = b + z, \text{ for } b \in B\} \tag{3}$$

where $(B)_z$ is the set of points c resulting from adding z to the points b from set B. If B is a set of 2D points in the format (x, y), then $(B)_z$ is the set of translated 2D points $(x + z_1, y + z_2)$. This facilitates the sliding window behavior that allows the SE to iteratively move across an object and operate on it. Similar to reflection, (3) translates objects with respect to their origin. Translation and reflection provide the building blocks to many other powerful morphological operations [32].

2.4.2 Erosion

Erosion degrades the foreground pixels of an object into background pixels if the SE fits into them. Given the sets A and B in 2D integer space, the erosion of A by B is defined as

$$A \ominus B = \{z | (B)_z \subseteq A\} \tag{4}$$

where A is a set of foreground pixels describing an object, B is a SE, and the z points are all foreground pixels. This means that an object A eroded by a structuring element B results in all points z such that B is contained entirely in A when translated by z. As mentioned earlier when defining translation, z refers to the origin of the translated SE [32].

With foreground pixels colored blue and background pixels colored white, Figure 5 depicts a 3x3 structuring element B with only foreground pixels that erodes object A. To make sense of this image, label the top left pixel of image I as $(0,0)_I$ and the top left pixel of the structuring element B as $(0,0)_B$. Consider the first translation of B to point $z = (1,1)$. Since objects translate in relation to their origin, this places $(0,0)_B$ on top of $(0,0)_I$ as annotated by the red box. Not all of the red box falls on foreground pixels of object A, so this operation does not add $z = (1,1)$ to the erosion set. Now consider the translation of B to point $z = (2,2)$ which results in the translation of B to the blue box. All foreground pixels in the structuring element intersect with the foreground pixels of A, so this operation adds $z = (2,2)$ to the erosion set. This results in the first foreground pixel seen at (2,2) in the image after the morphological operation. This process continues until B has translated across the entire image I to produce the eroded image.

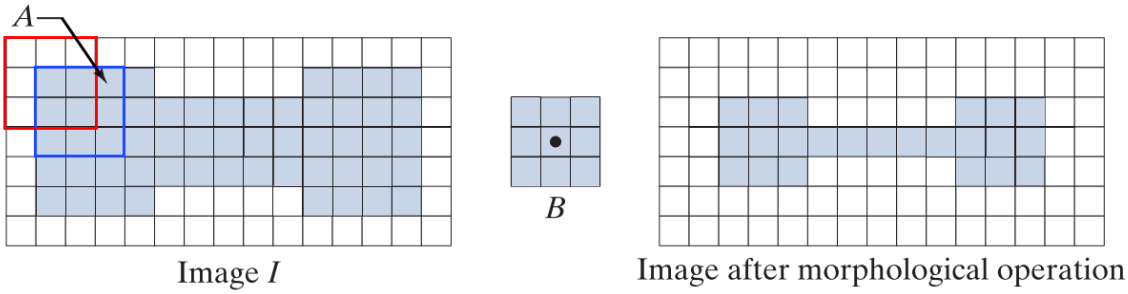

Figure 5: Object A eroded by 3x3 structuring element B [32]

2.4.3 Dilation

The opposite of erosion is dilation which expands the boundaries of an object by converting background pixels that touch it into foreground pixels. Given the sets A and B in 2D integer space, the dilation of A by B is defined as

$$A \oplus B = \{z|(\hat{B})_z \cap A \neq \varnothing\} \tag{5}$$

where A is a set of foreground pixels describing an object, B is a SE, and the z points are all foreground pixels. This means that an object A dilated by a structuring element B results in all points z such that B intersects with at least one foreground pixel in A when translated by z. Similar to erosion, the structuring element can contain background pixels which affects the end result [32].

Given a square object A with width d and a square structuring element B with width $d/4$, Figure 6 shows how A is dilated by B. Note the original perimeter of A marked by a dashed line on the dilated object. The center of a SE with width $d/4$ can only be up to $d/8$ away from the original perimeter while still touching the object which results in the marked expansion of the perimeter by $d/8$ on all sides.

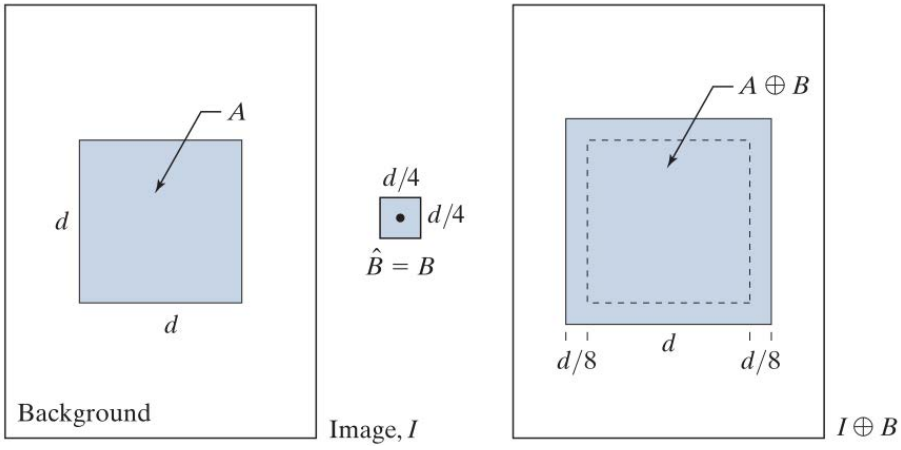

Figure 6: Object A dilated by structuring element B [32]

The definitions of reflection and translation facilitated the definitions of erosion and dilation. Now, the definitions of erosion and dilation can be used to define more advanced morphological operations.

2.4.4 Opening

Opening uses erosion and dilation to smooth contours and remove small objects and object protrusions. Given the sets A and B in 2D integer space, the opening of A by B is defined as

$$A \circ B = (A \ominus B) \oplus B \tag{6}$$

This operation first erodes the object A by B then dilates the result by B. Section 2.4.2 establishes that erosion only adds foreground pixels to the resultant set if the SE intersects exclusively with object foreground pixels. Therefore, the erosion operation erases any objects smaller than the SE after which the erosion operation returns the remaining objects to approximately their previous size [32].

2.4.5 Closing

Closing uses dilation and erosion to smooth contours on an object and fill small gaps in an object. Given the sets A and B in 2D integer space, the closing of A by B is defined as

$$A \bullet B = (A \oplus B) \ominus B \tag{7}$$

This operation first dilates the object A by B then erodes the result by B. Section 2.4.3 establishes that dilation adds foreground pixels to the resultant set if the SE intersects with any object foreground pixels. Therefore, the dilation operation fills any gaps smaller than the SE after which the erosion operation returns the objects to approximately their previous size [32].

2.4.6 Operations in Grayscale

Similar to binary images, all of the aforementioned morphological operations also work on grayscale images with a few adjustments. One notable difference is the ability to utilize a non-flat SE which has non-uniform intensity values for each pixel which complicates the operations. However, the operations can also use a flat SE on grayscale images which is the same format of SE used on binary images. The use of a flat SE keeps the operations relatively simple [32].

Reflection and translation do not use the pixel values, so there is no difference between the use on binary and grayscale images. Further, the definition for opening does not change since it just combines the erosion and dilation operations. However, the erosion and dilation operations do depend on the pixel values, so they require new definitions that account for grayscale images. Given a grayscale image f and a flat structuring element b, the erosion of f by b is defined as

$$[f \ominus b](x, y) = \min_{(s,t)\in b} f(x + s, y + t) \tag{8}$$

such that the erosion of a grayscale image by a SE centered at (x, y) results in the pixel at (x, y) taking on the minimum intensity of the pixels in the region covered by the SE. Similarly, the dilation of a grayscale image f by a flat structuring element b is defined as

$$[f \oplus b](x, y) = \max_{(s,t)\in \hat{b}} f(x - s, y - t) \tag{9}$$

such that the dilation of a grayscale image by a SE centered at (x, y) results in the pixel at (x, y) taking on the maximum intensity of the pixels in the region covered by the reflected SE. Recall that reflection results in $\hat{b}(x, y) = (-x, -y)$ which changes the sign of s and t [32].

2.4.7 Tophat

Section 2.4.4 establishes that the erosion step in an opening operation erases objects smaller than the SE. Therefore, subtracting the result of the opening from the original image results in an image containing only objects smaller than the SE used in the erosion step of the opening. This process is called a tophat transformation. Given a grayscale image f and a structuring element b, the tophat transformation of f is defined as

$$T_{hat}(f) = f - (f \circ b) \tag{10}$$

The opening operation results in a grayscale image which is subtracted from the original grayscale image. To subtract a grayscale image from another of the same size, the intensity of each pixel is subtracted from the intensity of the corresponding pixel on the other image. If the subtraction results in an intensity less than 0, the intensity is set to 0 [32].

2.4.8 Gaussian Blur

Similar to morphological operations, the Gaussian Blur operation refers to the use of a weighted average spatial filter that samples its weights from a Gaussian function to smooth an image. Rather than a structuring element, this operation uses a filter called a kernel which slides across the image much like a structuring element. This kernel multiplies the pixel intensities by its weights and averages the values to set the intensity of the current pixel at its center. Given a kernel w, the operation $w(x, y)$ stores a real, discrete value at location (x, y). A Gaussian kernel samples values from a 2D Gaussian function of the form

$$w(x, y) = G(x, y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}} \tag{11}$$

where σ is the standard deviation of the Gaussian distribution. Gaussian kernels can be used to smooth noisy images as well as blur sharp images. In general, a larger kernel size and higher standard deviation results in a blurrier image [32].

2.4.9 Affine Transformation

An affine transformation is a 3D transformation of a 2D image that preserves points, straight lines, and planes. Affine transformations have four categories: scaling, translation, rotation, and shearing. All four transformations can be expressed in the general form

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = A \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \tag{12}$$

where A represents the transformation matrix that transforms coordinates (x, y) to (x', y'). The translations are critical for reproducing the behavior of 2D planes projected into 3D space [32].

2.4.10 Image Thresholding

Image thresholding refers to the process of identifying an appropriate pixel intensity value to act as a boundary for partitioning a grayscale image. Given a threshold value T, the intensity of a pixel at location (x, y) in a thresholded image g is defined as

$$g(x, y) = \begin{cases} 1 & \text{if } f(x, y) > T \\ 0 & \text{if } f(x, y) \leq T \end{cases} \tag{13}$$

(13) shows how image thresholding converts grayscale pixel intensities to binary pixel values [32].

Image thresholding algorithms operate on images globally or locally. Global al-

gorithms look at the pixel intensity of every pixel in an image to determine a single threshold value while local algorithms analyze smaller sections of the image and determine threshold values for each local section. Regardless of whether an algorithm is global or local, most algorithms analyze the variance of pixel intensities to identify a threshold value [32].

A histogram of grayscale intensities in an image can show multiple distinct peaks if there is a stark enough contrast between intensities in an image like in Figure 7. Using the intensity value from the valley between the peaks often results in a good threshold value, but this process is also highly affected by noise, lighting, and material reflectance. Even in high quality photos, the separation of peaks in the histogram might not indicate such an obvious thresholding value. Therefore, a more thorough approach is required [32].

The most popular and highly effective global thresholding algorithm is Otsu's Method [34]. Given an image with L distinct pixel intensities, there are k possible threshold values such that $0 < k < L - 1$. Each k splits the intensities into two classes: $c_1 = [0, k]$ and $c_2 = [k + 1, L - 1]$. If the mean of both classes differs from the global mean significantly, then the selected k likely thresholds the image well. To prevent high variant, low frequency intensities from skewing this comparison, Otsu's Method calculates a between-class variance. With p_i as the number of pixels with intensity i, this method defines

$$P_1(k) = \sum_{i=0}^{k} p_i \tag{14}$$

as the probability that a pixel is assigned to c_1. This method then defines

$$P_2(k) = \sum_{i=k+1}^{L-1} p_i = 1 - P_1(k) \tag{15}$$

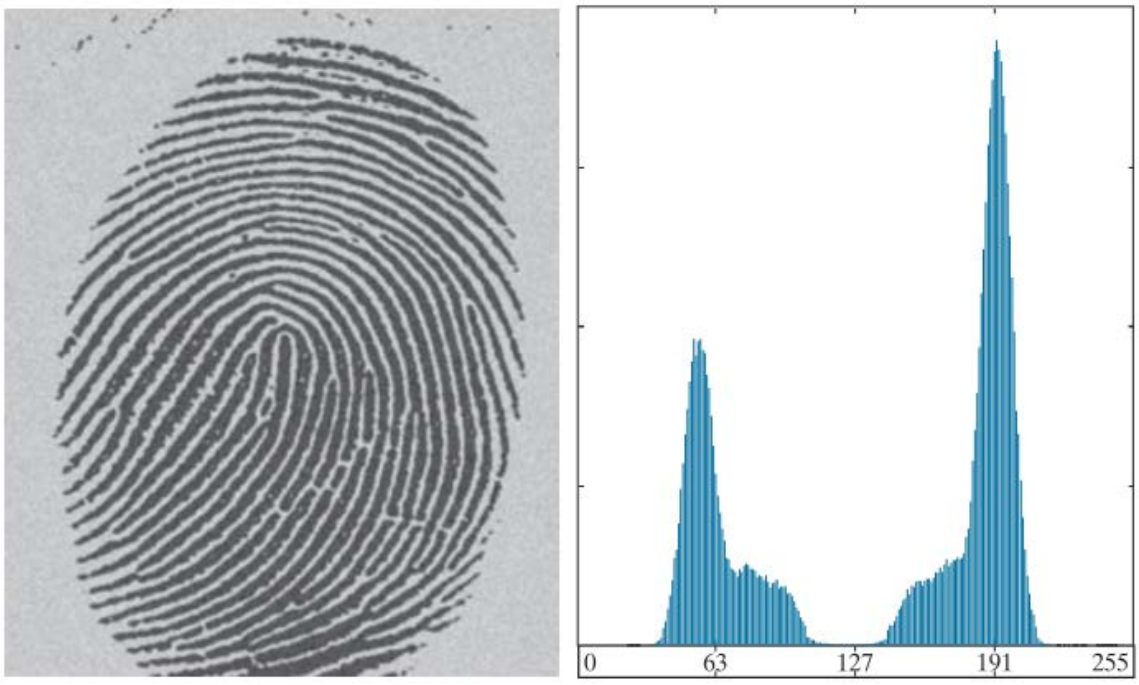

Figure 7: Intensity histogram of grayscale image [32]

as the probability a pixel is assigned to c_2. Otsu's Method uses these probabilities to define the mean intensities of c_1 in (16), the mean intensities of c_2 in (17), and the average global intensity in (18).

$$m_1(k) = \frac{1}{P_1(k)} \sum_{i=0}^{k} ip_i \tag{16}$$

$$m_2(k) = \frac{1}{P_2(k)} \sum_{i=k+1}^{L-1} ip_i \tag{17}$$

$$m_G(k) = \sum_{i=0}^{L-1} ip_i \tag{18}$$

With these probabilities, Otsu's Method then defines between-class variance as

$$\sigma_B^2 = P_1(m_1 - m_G)^2 + P_2(m_2 - m_G)^2 \tag{19}$$

Otsu's Method maximizes this variance between classes to find the optimal threshold value.

$$\sigma_B^2(k*) = \max_{0 \leq k \leq L-1} \sigma_B^2(k) \tag{20}$$

Upon finding the maximum value for σ_B^2 with (20), the maximizing $k*$ is returned as the optimal threshold value [32].

This optimal threshold value is then used as the threshold value T in (13) to produce a binary image that isolates the high frequency intensities in an image. Figure 8 shows how Otsu's Method isolates cellphone LiDAR light points in a dark image.

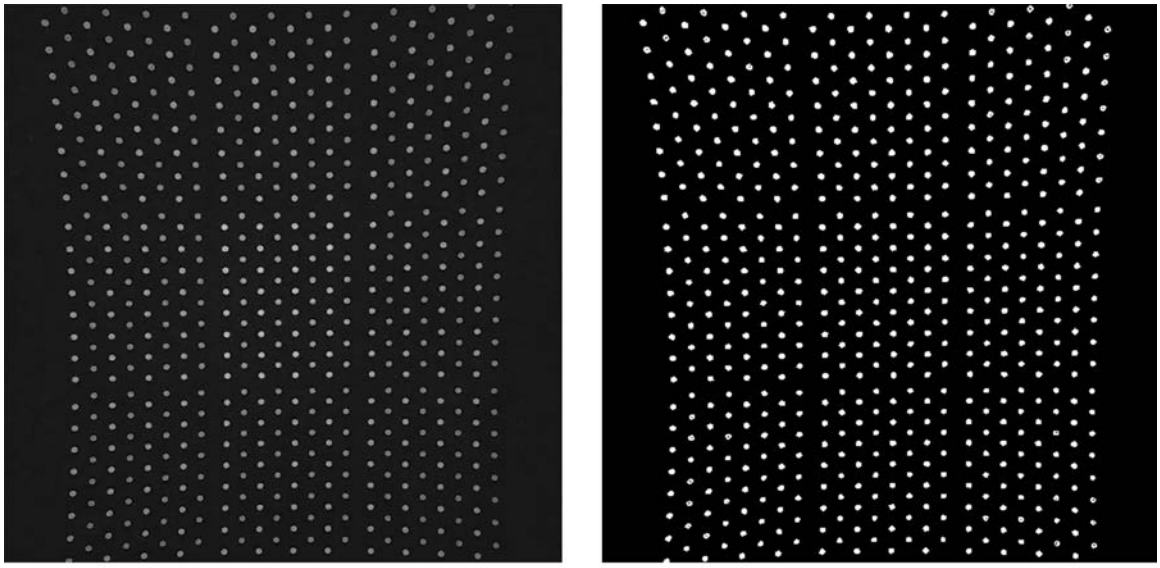

Figure 8: Otsu's Method thresholds cellphone LiDAR light image (a) into masked image (b)

2.4.11 Background Subtraction

Background subtraction refers to the operation that subtracts a baseline frame from another frame in a video. The simplest form of subtracting a saved frame works in very limited cases, but overall it is extremely sensitive to environmental changes

like lighting. More advanced methods use Gaussian mixture models to improve performance. Rather than subtracting a single frame from every other frame, the model tracks and averages past frames so it can subtract a more accurate representation of the environment from future frames. Many of these implementations require color video which degrades their performance on videos that capture NIR light since they must record in grayscale. However, other implementations of the Gaussian mixed models perform well on grayscale videos [35].

The two most notable implementations are K-Nearest Neighbors (KNN) and Mixture of Gaussians (MOG). KNN uses recursive update equations that keep track of past frames while MOG improves the original update equations used in the Gaussian mixture models. MOG performs best on color videos, but both KNN and MOG can effectively detect movement in grayscale videos. Unfortunately, both of these algorithms require manually tweaking a threshold value to ensure the proper amount of light is subtracted from a frame given the current lighting [35].

2.4.12 Fourier Transform

A Fourier transform converts a signal from its current domain, usually time or spatial, into the frequency domain. The decomposed frequencies can often represent useful information that is not easily observable in the original signal such as noise patterns [32]. In similar fashion, a 2D Discrete Fourier Transform (DFT) converts the 2D signal from its domain into the frequency domain with the equation

$$F(u,v) = \sum_{x=0}^{A-1} \sum_{y=0}^{B-1} f(x,y) e^{-j2\pi(ux/A+vy/B)} \tag{21}$$

where $f(x,y)$ denotes the intensity of a pixel located at (x,y) in an image with height A and width B. This returns both real and imaginary number components which can

be combined into a single magnitude value for each pixel with

$$M(u,v) = \sqrt{Real(F(u,v))^2 + Imag(F(u,v))^2} \tag{22}$$

where $Real()$ and $Imag()$ represent the separate real and imaginary components of a single 2D DFT [32].

When applied to a grayscale image, this transform can identify visual patterns in the image based on the frequency of pixel intensities at various locations [11]. Although the 2D DFT typically produces large magnitude values, the values can be scaled to visualize it. Figure 9 shows the 2D DFT of the LiDAR light image thresholded by Otsu's Method in Figure 8(b). The resulting 2D DFT image displays a rather unique pattern that can be used to identify it as LiDAR light with a frequency pattern classifier implemented in a machine learning neural network [11].

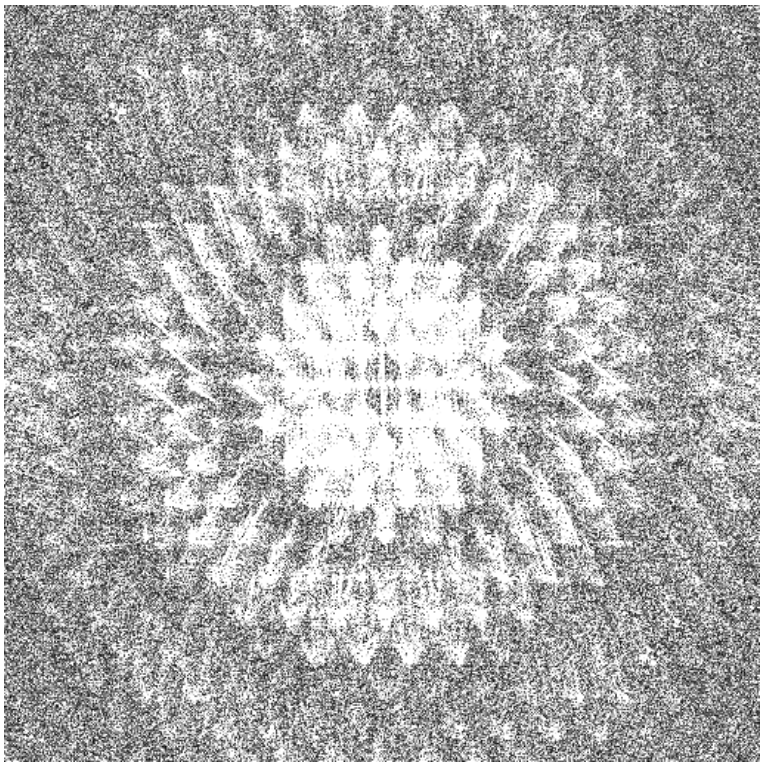

Figure 9: 2D DFT of Otsu's Method result

2.5 Machine Learning Image Classification

Machine learning refers to the ability of a computer to acquire knowledge by extracting patterns from raw data rather than relying on hard-coded knowledge [36]. Although machine learning applications are able to acquire new knowledge from raw data, it expects to receive the raw data in a particular format called a representation. An approach called 'representation learning' attempts to learn how the represented data influences the output as well as the format of the representation itself. This may be a rather complex task for which manually designing a model takes too much time or effort. In lieu of manually designing a model, deep learning accomplishes this by introducing simpler representation which can combine into more complex representations. For example, a deep learning model may represent a human as a collection of simpler concepts such as corners and contours rather than a singular human shape [36].

Deep neural networks conceptually operate more complexly than basic machine learning networks, but they are still neural networks at the core. A neural network emulates how neurons in a human brain receive signals and activate upon certain conditions to pass signals to other connected neurons. A basic neural network consists of three main parts: the input layer, the hidden layers, and the output layer [36].

The input layer of a neural network ingests data to pass onto the first hidden layer of the network. In the case of image processing, the input layer properly formats the image into an array or tensor for the hidden layers to operate on. Each hidden layer consists of neurons that activate based on the specifics of their activation functions. These activation functions may be linear or non-linear, but a common linear activation function called the Rectified Linear Unit (ReLU) is defined as

$$f(x) = max(0, x) \tag{23}$$

where x is a real, numerical value.

A ReLU neuron simply passes along the value it receives if it is positive, otherwise it passes along a 0. Neurons often apply weights and biases before activation which slightly modifies the function to

$$f(x) = max(0, w * x + b) \tag{24}$$

where w is a real, numerical weight value and b is a real, numerical bias value [36].

When the hidden layers pass their values to the output layer, the output layer must use the aggregated data to make a decision. In an object classification neural network, the output layer decides to which class the input belongs. The output layer contains one neuron for each possible class that uses the softmax function as its activation function. Given an array x with n elements, the softmax function is defined as

$$\text{softmax}(x)_i = \frac{\exp(x_i)}{\sum_{j=1}^{n} \exp(x_j)} \tag{25}$$

where x_i is the ith element of x and x_j is the jth element of x [36]. Applying (25) to each element of x results in the array softmax(x) where the ith element contains the result of softmax$(x)_i$. This normalizes the values to the range $0 <$ softmax$(x)_i < 1$ such that they sum to 1. The output layer then decides which class the input belongs to based on the index of the maximum softmax$(x)_i$. If the maximum value of softmax(x) is at index 3, then the input belongs to class 3.

The softmax function is a generalization of the binary classification sigmoid function which is defined as

$$\sigma(x) = \frac{1}{1 + \exp(-x)} \tag{26}$$

where x is the real, numerical value passed to the output layer [36]. Both the softmax and the sigmoid function calculate the probability that the input belongs to each

class which always sums to one. However, a softmax output layer requires one output neuron for each class whereas a sigmoid output layer only requires one neuron total since the decision is binary. Regardless of the activation function, the output layer returns the class with the highest calculated probability as the answer. The accuracy of the answer can be used to update the weights and biases of the hidden layers to achieve more accurate results [36].

The output most commonly feeds back updates to the hidden layers' trainable variables (weights and biases) through the process of Stochastic Gradient Decent (SGD). SGD subtracts the gradient of the loss function from trainable variables to slowly improve towards a favorable output. A loss function measures the inaccuracy of the output layer result. In binary classification, the loss function is the binary cross entropy function which calculates the average of the log of predicted probabilities as seen in

$$\text{Loss} = \frac{1}{N}\sum_{i=1}^{N} log(p_i) \tag{27}$$

where N is the number of inputs and p_i is the probability with which the input belongs to a target class as calculated by the output layer [36]. The target class is one of the two possible classes in a binary classification neural network. The chosen class is arbitrary but must remain consistent.

(27) calculates loss from the probabilities that the sigmoid function calculates from values passed through the hidden layers. This means all trainable variables in the hidden layers affect the loss function, so SGD updates each trainable variable by subtracting the partial derivative of the loss function with respect to that variable. Intuitively, the partial derivative of the loss function with respect to a trainable variable represents the rate at which that variable affects the loss. In order to minimize loss, subtracting the result of that derivative lowers the impact of the variable if the variable increases loss and increases the impact of the variable if the variable decreases

loss [36].

Updating a trainable variable may overshoot if the partial derivative value subtracts or adds too much. It is common to multiply the partial derivative by a small step value called the learning rate to control how aggressively the trainable variables update [36].

$$v_{t+1} = v_t - r\frac{\partial Loss}{\partial v} \tag{28}$$

Putting this all together, (28) shows how a trainable variable v updates during the training process [36]. The equation first calculates the partial derivative of the *Loss* function from (27) with respect to the trainable variable. It then multiplies this partial derivative by the learning rate r which it subtracts from the current value of the trainable variable v_t to obtain the updated value v_{t+1}. This is the generic SGD process that various other algorithms improve on such as the popular algorithm Adam [37]. Although research shows that SGD generally performs better, Adam is shown to perform better in particular circumstances [38].

Training a neural network typically requires multiple epochs. An epoch completes when the neural network processes each piece of training data once. Each epoch consists of multiple steps. A step completes when the neural network processes one batch of training data. A batch consists of at least 1 piece of training data and is limited only by computing resources [36].

2.5.1 Frequency Pattern Classifier

As described in Section 2.5, deep neural networks decompose the representation problem into a collection of simpler representation. Section 2.4.12 also notes that Fourier transforms decompose signals into simpler frequency components. This makes the Fourier transform a good candidate for extracting features from an image to clas-

sify it. When applied as the final step of a deep neural network, a 2D DFT transform proves to be a highly effective way to classify patterns with increased efficiency over traditional image classification neural networks. The following paragraphs elaborate further on this methodology proposed by Stuchi et al. in 2020 [11].

Figure 10 shows the general neural network design. The design contain five main steps: recursively slice the layers into pixel blocks, perform a 2D DFT on each pixel block and calculate its magnitude block, pool the magnitudes into rings, multiply each ring by its own trainable filter weight, then process weighted values through a neural network [11].

The process of slicing the image results in subsections of a single image. It is highly preferable that the image is square for ease of computation. The image is sliced into 4 equal squares, and each of those are sliced into 4 more equal squares resulting in 21 total images. The images are not resized, so each slice results in smaller images [11].

As described in Section 2.4.12, a 2D DFT applied to each image produces real and imaginary numbers that must be combined into magnitude values. The resulting magnitudes still represent the DFT frequency domain that contains high frequencies on the border and low frequencies in the center. Since frequencies group together radially in the frequency domain, pooling them with a radial filter enables the emphasis

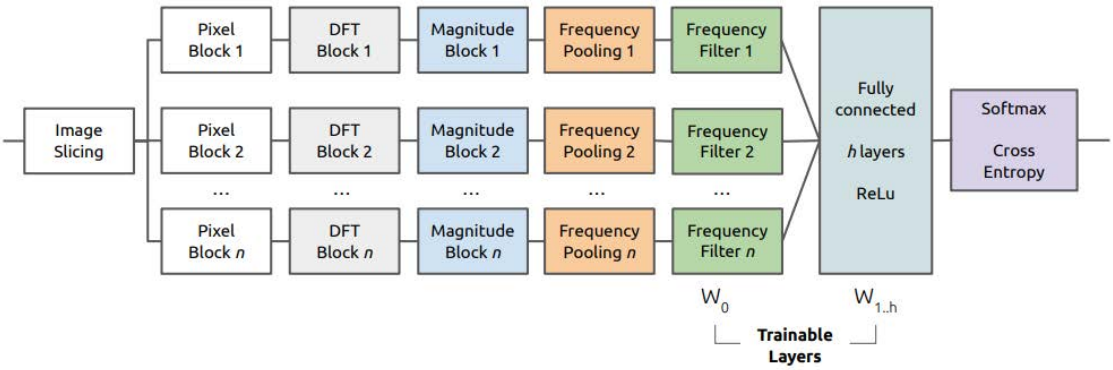

Figure 10: Basic frequency layer architecture [11]

of certain frequencies when multiplied by their respective weights [11].

Band pass filters commonly use Euclidean distance to calculate radial filters, but [11] instead uses the Chebyshev distance to calculate a square filter as seen in Figure 11. The rings of the filter may be wider than 1 pixel, but all rings must be the same width. This step creates a filter for each DFT magnitude block which results in 21 total filters [11]. The Chebyshev distance is defined as

$$D(x, y) = max(|x_2 - x_1|, |y_2 - y_1|) \tag{29}$$

where $D(x, y)$ represents the Chebyshev distance between two pixels located at $x = (x_1, x_2)$ and $y = (y_1, y_2)$ respectively [32]. Given an image with ring width R_w, the equation

$$Ring(x) = floor(D(x, mid)/R_w) \tag{30}$$

calculates the ring number $Ring(x)$ that pixel $x = (x_1, y_1)$ belongs to based on its distance from the middle pixel $mid = (x_2, y_2)$ of the image. The floor operation converts the decimal result of the division operation to the nearest integer less than or equal to the decimal. This facilitates frequency pooling for rings wider than one pixel.

Similar to how hidden layer neurons have trainable weights, each ring in each filter has one trainable weight. The intensity of every pixel in a ring is summed to a single ring value which is multiplied by its respective ring weight. The results of this product are stacked up and sent to the hidden layers as a combined input value for one image [11].

This greatly reduces the number of required trainable variables. Instead of training 1,048,576 individual weights to multiply each pixel of a 1024 x 1024 grayscale image, the image can be pooled in 512 rings that only require 512 trainable variables.

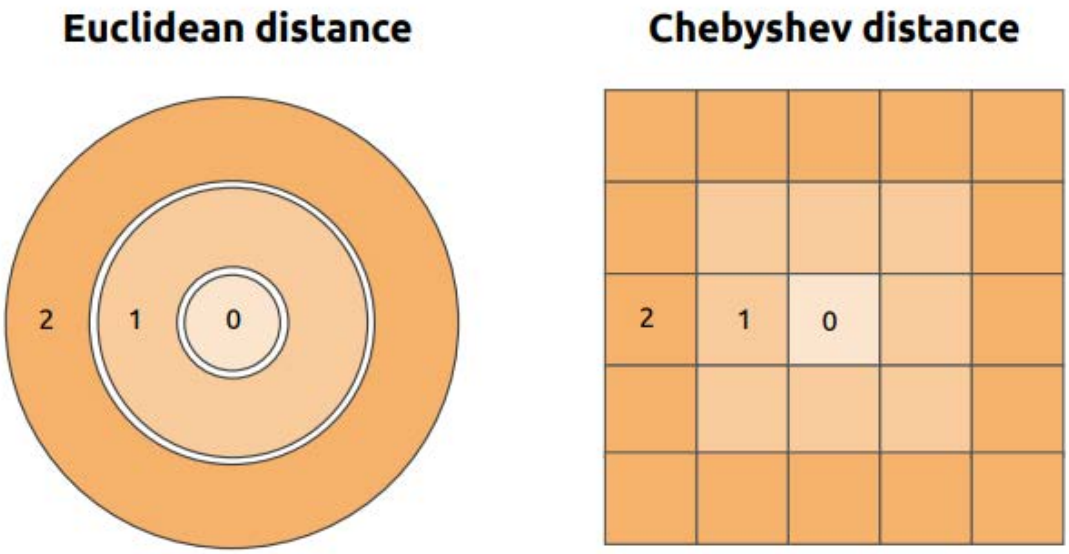

Figure 11: Euclidean vs. Chebyshev distance filter [11]

Furthermore, the operations used sum and multiply the rings by the respective ring weights are differentiable which allows the SGD process described in Section 2.5 to update the ring weights based on the loss function as it trains [11].

2.6 Tools

The tools used in this research support three primary environments: Python development, Apple development, and cellphones. The cellphone environments consist of both the cellphone hardware and applications.

2.6.1 Python Development Environment

Python is a high-level interpreted programming language that provides access to powerful libraries and packages. Python has two major versions: Python 2 and Python 3. Python 2 is slowly disappearing in favor of Python 3 [39]. Development of code in Python 3 is facilitated by PyCharm.

PyCharm is a full fledged Python Integrated Development Environment (IDE)

that allows developers to write, run, debug, and test Python code all in one program. It provides many excellent features for free such as GitHub version control, best practice code formatting, and even management of Conda environments [40].

Conda environments isolate installed Python packages so separate projects do not need to share one global directory of packages. Conda also provides access to some packages that only exist in the Conda installation archives rather than the conventional pip archives [41].

Pip is the native Python package installer program that installs packages from the pip archives to a global directory. Combining this with Conda allows developers to use their favorite pip packages in isolated environments [42]. Conda typically manages packages better, so pip is a last resort for installing a required package if Conda cannot install it.

The primary packages used in this research are OpenCV, TensorFlow, NumPy, and Scikit-Image. Section 3.5 provides further specifications on these packages.

OpenCV is an open-source computer vision library that provides the functions for the majority of the image processing completed in this research as well as the functions for recording video footage from the security camera. Although it is written in the programming language C++, the Python package provides a front end with access to all of the functions available in C++ [43].

TensorFlow is an open-source machine learning and artificial intelligence library. Although other libraries provide the necessary functions to build and train custom deep neural networks, TensorFlow provides many unique functions such as automatic differentiation that trivializes the implementation of SGD and Adam. The main data unit in TensorFlow is the tensor; this is an immutable, multi-dimensional array of data that also stores metadata such as tensor shape. In many cases, these tensors can be converted to NumPy matrices for easier handling [44].

NumPy provides a host of useful mathematical functions, but it most notably provides a robust, easy to use matrix platform. The matrices support many advanced mathematical operations, but they are also compatible with most TensorFlow functions [45].

Scikit-Image is an open-source Python library that provides many image processing functions. Although OpenCV provides a slew of image processing functions, there are a few that only Scikit-Image provides [46].

Imutils provides quality-of-life improvement wrappers for OpenCV. Most notably, this library makes it easier to rotate images [47].

Other minor packages used in this research are Python standard libraries. These include 'os' which provides operating system file management functions, 'math' which provides mathematic functions, 'pathlib' which provides file directory navigation functions, 'random' which provides randomised sampling functions, and 'shutil' which provides directory and file deletion functions. This also include the 'time' library which provides time functions that measure how long it takes Python scripts to execute with nanosecond (ns) accuracy. [48].

The experiments in this research evaluate the classification accuracy of a TensorFlow neural network and the total run time of different Python scripts. Since the Python 'time' library measures to nanosecond accuracy where 1 nanosecond is 10^{-9} seconds, the run time results are only accurate to 9 decimal places. TensorFlow reports classification accuracy of the trained neural network up to 16 decimal places, so the accuracy results are only accurate to 16 decimal places. Therefore, any calculations that use the run time or accuracy cannot produce results more precise than 9 or 16 decimal places respectively. NumPy includes functions that calculate mean and standard deviation with the same precision as the input data. Therefore, the means and standard deviations of the classification accuracy have 16 decimal places

of precision, and the means and standard deviations of the run time have 9 decimal places of precision.

NACL Web Plug-in is a Google Chrome extension that facilitates connection to a security camera to view the live video stream as well as change the camera settings. This program is not related to Python, but it is required in order to change the camera settings since Python can only record video from the security camera, not change the settings [49].

2.6.2 Apple Development Environment

Apple computer and phone apps can be written in a programming language called Swift. Swift is a general-purpose compiled programming language supported by a large open source community. Although Swift is cross platform compatible with Linux, it is arguably easiest to develop programs in Swift on an Apple computer in Xcode [50].

Similarly to PyCharm, Xcode is an IDE made by Apple specifically to develop software for Apple devices on Apple computers. This program provides many convenient development features such as the ability to push custom apps to Apple phones and launch them without having to publish them publicly [51].

2.6.3 Cellphones

Cellphones embedded with LiDAR sensors are a fairly recent advent. Although more companies are producing LiDAR enabled phones as time passes, three well known companies produce cellphones with rear LiDAR sensors: Apple, Samsung, and Huawei. Apple produces the iPhone 12 Pro Max [52], Samsung produces the Galaxy S20 Ultra 5G [53], and Huawei produces the P40 Pro [54]. All three of these devices possess rear Time of Flight LiDAR sensors [7][55][56]. Applications available

on their phones' respective app stores as well as on the Internet facilitate use of these LiDAR sensors.

2.6.4 Cellphone Applications

The various hardware of cellphones is primarily utilized through applications. Cellphones download these applications through an application ("app") store that whitelists applications users may use. Samsung phones use the Google Play store [57] while Apple and Huawei phones have their own Apple and Huawei app stores respectively [58][59]. Although the phones cannot use apps from each others' app stores, the app stores often have similar if not identical apps available on them. The Google Play and Huawei store have the same 3D Live Scanner app available for download [6][60], as well as many other 3D scanning apps. The Apple app store also has many 3D scanning apps available, but Apple also has the source code of a 3D LiDAR scanning app available on their developer website that can be directly loaded onto an iPhone from a Mac computer instead of downloading it from the Apple app store [61]. All three app stores also posses a variety of free light meter apps, but research shows that the Photone light meter app available on the iPhone app store performs the best among many other light meter apps [62].

2.7 Related Research

This research consists primarily of recording LiDAR light with a security camera and processing the video frames to identify the presence of LiDAR light with a neural network. These two parts consist of the following research topics: capturing LiDAR light with a CMOS camera, emphasizing and isolating high intensity pixels in an image, and classifying a pattern in an image.

2.7.1 LiDAR Light Capture with CMOS Camera

The development of LiDAR has progressed drastically since its creation in the 1960s, seeing its use in various consumer and defense applications across the globe [4][5][63]. Although many 1D LiDAR solutions exist, 3D LiDAR also has a myriad of environmental, defense, and consumer applications [64]. Many companies have pursued research in long range 3D LiDAR defense projects [65][66][67]. The Defense Advanced Research Projects Agency (DARPA) developed the Standoff Precision Identification in Three Dimensions (SPI-3D) project to identify targets at standoff range with LiDAR and the Jigsaw program to detect ground forces through camouflage and foliage with long range aerial 3D LiDAR [68]. Similarly to the cellphone LiDAR sensors this research concerns, other recent COTS 3D LiDAR technology proves applicable in many consumer and defense projects [69][1]. This long history of development of not just LiDAR, but 3D LiDAR, shows the defensive and offensive capabilities of 3D LiDAR. This research develops a methodology to detect and identify cellphone 3D LiDAR with a COTS CMOS camera that may potentially extend to detect and identify adversarial use of these other defensive and offensive 3D LiDAR capabilities.

Since the wavelength of LiDAR light falls in the NIR spectrum, the ability to detect LiDAR light with a CMOS camera relies on a CMOS camera's ability to capture NIR light. Much research explores infrared imaging with CMOS cameras for applications such as industrial temperature sensing, telescopes, and medical imaging [70][71]. Research also details the use of CMOS cameras for NIR imaging applications such as the impact of handling food and crop analysis [72][73][74]. Unfortunately, the majority of these applications use expensive, specialized CMOS cameras. Much research exists on capturing NIR light with inexpensive, general purpose CMOS cameras for applications such as detecting pedestrians and analyzing plants [75][76][77]. However, many of these applications implement a high-pass NIR filter that filters out

visible light in order to isolate the NIR light in captured footage. Fortunately, some research uses unfiltered, inexpensive, general purpose CMOS cameras that capture visible and NIR light for purposes such as artwork monitoring, fuel characterization, and vegetation monitoring [78][79][80]. Among these various studies, results indicate that various inexpensive COTS cameras, including security cameras, are capable of capturing NIR light in video footage which can then be processed to extract different characteristics of the NIR light.

2.7.2 Emphasizing and Isolating High Intensity Pixels

Once NIR light is captured, emphasizing it in a given image makes it easier to isolate. The method chosen to isolate the NIR light influences how the light is emphasized, so this section first discusses NIR light isolation.

Isolating NIR light from an image is similar to background subtraction, but background subtraction refers to a specific class of algorithms as described in Section 2.4.11. Both the KNN and MOG background subtraction methods can subtract background pixels to isolate NIR light in certain circumstances [35]. However, other algorithm classes can also isolate NIR light if properly emphasized. Small target tracking algorithms efficiently isolate small points in infrared images that may be potential targets and classify which ones are likely targets [81][82][83]. The small points do not need to be high contrast compared to the background, but the background must contain fairly consistent patterns and intensities across the image. This consistency cannot be assumed for LiDAR light projected onto various backgrounds. However, these algorithms do indicate certain morphological operations that can emphasize small points such as Gaussian blur, dilation, and closing. Segmentation methods developed for infrared medical imagery also use morphological operations to isolate structures, but they first emphasize the structures.

Methodologies for isolating structures such as hands in infrared medical imagery often requires the structures to first be emphasized. Although the exact process varies based on the structure, the methodologies heavily rely on morphological operations to emphasize the structure [71][84][85]. These morphological operations commonly include closing to connect components and opening to remove small components. Once the structure is properly emphasized, these methodologies then use image thresholding such as Otsu's Method to isolate the structure from the background which results in a binary image displaying the white foreground structure on a black background.

2.7.3 Pattern Classification

Once the NIR light is emphasized and isolated, it can finally be classified. Neural networks can be leveraged to classify color images such as hyperspectral images, coal images, and medical images [86][87][88]. In absence of color images, other neural networks classify grayscale images accurately [89][90][91]. Whether color or grayscale, both of these categories of neural networks often rely on convolution which recursively dilutes the image to extract context information. This is a well known, useful technique, but it requires an image with suitable context information. This context information often exists as a gradient of pixel colors or intensities in color or grayscale images respectively. Binary black and white images lack this context information, so convolution does not work well on them. Other techniques treat images as textures rather than objects and classify them based on general patterns rather than object characteristics [92]. This idea can be implemented in a neural network by training it on the pixel intensity frequencies derived from a 2D DFT [11]. This allows a neural network to classify an image by detecting what texture pattern it is most similar to based on the frequency and location of pixel intensities. Not only does this method perform well on binary black and white images, but its neural network training time is

also orders of magnitude faster than comparable image classification neural networks [11][90][91].

2.7.4 Related Research Summary

This research faces many limitations in the steps of detecting and identifying LiDAR light in security camera footage. However, the literature review details the applicable pieces of information derived from related research that facilitate the goal of this research when aggregated. Table 1 summarizes the relevant parts of referenced research as well as the categories this research fulfills. The **Camera** category refers to the use of an unfiltered, inexpensive, general purpose COTS camera. The **Emphasize** category refers to the use of morphological operations to emphasize NIR/IR light in an image. The **Isolate** category covers the process of isolating desired NIR/IR light in an image. Lastly, the **Classify** category marks the research that uses a neural network to classify NIR/IR light in an image as belonging to a certain class.

Table 1: Related research

Authors	**Camera**	**Emphasize**	**Isolate**	**Classify**
Dantas et al. (2017) [79]	X			
Petach et al. (2014) [80]	X		X	
Trnovszky et al. (2017) [35]			X	
Ju et al. (2019) [81]			X	X
Herry et al. (2006) [71]		X	X	
Stuchi et al. (2020) [11]				X
This Research	X	X	X	X

2.8 Background Summary

This chapter elaborates on the fundamental LiDAR principles and how they relate to the characteristics of CMOS cameras. Image processing techniques are then discussed followed by information on what neural networks are and how they can

be used to detect patterns in images. The chapter concludes with the tools used to support this research as well as a discussion of related research.

III. System Design

3.1 Overview

This chapter presents the design of a system to detect and identify cellphone LiDAR light in security camera footage. Section 3.2 presents multiple design considerations that affect the final design of this system. Section 3.3 summarizes the system and the goals of each component within the system followed by a detailed description of each component and its implementation in Section 3.4. Section 3.5 lists the versions of the tools used to develop this research. Finally, Section 3.6 lists the specifications for the devices used to accomplish this research.

3.2 Design Considerations

The following sections explore concerns about the LiDAR hardware and image processing techniques that affect the system design. The problem of identifying LiDAR light from different cellphones is first discussed.

3.2.1 Cellphone LiDAR Design

As identified in Section 2.3.7, the indirect ToF LiDAR light projected in Figure 4 lacks unique characteristics that differentiate it from visible light. Among the three phones acquired for this research, both the Samsung Galaxy Ultra 5G and Huawei P40 Pro possess indirect ToF LiDAR sensors. Only the iPhone 12 Pro Max possesses a direct ToF LiDAR sensor. This limits the cellphones to just the iPhone 12 Pro Max which is hereby referred to as 'iPhone', 'the phone', or 'the cellphone'. Therefore, all mentions of 'LiDAR light' henceforth refer to the LiDAR light emitted from an iPhone 12 Pro Max rear LiDAR sensor. With this in mind, a process must now be identified to isolate only the light points of direct ToF LiDAR light in an image. It

may seem trivial to extract the light points seen in Figure 3, but an isolation method should be generally applicable in different lighting conditions.

3.2.2 Background Subtraction

Background subtraction algorithms prove near worthless for extracting the light points of direct ToF LiDAR light in an image. Rudimentary background subtraction that subtracts a baseline frame from the current frame is far too sensitive even in a controlled research environment. The KNN and MOG algorithms that make background subtraction more robust do not suffer from such sensitivity, but they quickly wash out stationary objects from the scene since they keep a running average of past frames. If an object remains stationary for multiple frames in a row, it is soon considered part of the background and subtracted out of the image. Even worse, they both contain threshold values that must be manually configured to appropriately subtract the background based on the lighting of the environment. Since rudimentary background subtraction is too sensitive and the more advanced algorithms wash away the LiDAR light points, this research develops a background subtraction method from a combination of morphological operations.

3.2.3 Morphological Operation Necessity

Before discussing the morphological operations, this section discusses their necessity. Recall from Section 2.5.1 that the classification algorithm classifies objects based on the intensity of every pixel in the provided image. This works well on images that consist mostly of target class. The target class dataset of this research unfortunately contains much unwanted empty space between the LiDAR light points that is filled by the surface it reflects off. This results in two potential options: design a dataset of the LiDAR light points laid over different surfaces and hope that covers a suitable

amount of use cases, or remove the background surface from the image and only leave behind the LiDAR light points. The latter option is preferable since it facilitates identification of LiDAR light points on any surface, but it is more difficult to accomplish since the aforementioned background subtraction algorithms fail to isolate the LiDAR light points. Therefore, this research combines morphological operations to isolate the light points instead.

3.2.4 Light Point Emphasis

Among the morphological operations described in Section 2.4.1, tophat shows the most promise for background subtraction to emphasize the light points. However, the erosion operation in the opening operation of tophat erases LiDAR light points smaller than the size of its Structuring Element. Although the opening operation may erase noise, subtracting it from the original image retains noise which makes it difficult to segment with Otsu's Method. Therefore, an alternative method is used to emphasize the LiDAR light points and improve segmentation by Otsu's Method.

Before performing other operations, a Gaussian blur filter removes noise so it does not affect subsequent operations. Next, closing the image smooths the contours of the light points and closes any gaps to make each light point solid. After closing the image, performing an additional dilation increases the size of the light points to emphasize them in the image. Performing both the closing and dilation operation with a circular kernel maintains the circular shape of the LiDAR light points. The Gaussian blur, closing, and dilation are all performed with a kernel size (5, 5) which this research finds to perform well, but further research is needed on the influence of each kernel size.

The final process for emphasizing the light points then is as follows:

- Gaussian blur with (5, 5) kernel

- Close with (5, 5) circular kernel
- Dilate with (5, 5) circular kernel

After the light points are emphasized, they can then be isolated with Otsu's Method.

3.2.5 Light Point Isolation

With the light points emphasized, they now just need to be isolated from an image. Section 2.4.10 describes the process of thresholding an image with Otsu's Method based on the intensity value that maximizes the between-class variance of pixel intensities. This thresholding process is a global operation that considers all pixel intensities at once whereas the morphological operations that emphasize the light points instead slide a kernel along the image and iteratively perform local operations. In images of LiDAR light points, this causes Otsu's Method to be influenced by high intensity pixels even if they are far away from the LiDAR light points. To prevent this, performing Otsu's Method across a sliding window similar to the morphological operations ensures that Otsu's Method looks at local pixel neighborhoods to decide local threshold values rather than a single global threshold value.

This process consists of sliding a window across the image, emphasizing the light points as detailed in Section 3.2.4, and thresholding the image with Otsu's Method as detailed in Section 2.4.10. Each processed window replaces the original window in the original image resulting in an entire processed image. This research refers to this light point isolation methodology as Sliding Window Otsu's Method (SWOM).

This research uses images of size 512 x 512 pixels based on the classification neural network research described in Section 2.5.1 Given an image of size 512 x 512, an optimal sliding window size must be decided for SWOM. The window must evenly divide the image, so a sliding window of format (size, size) has the potential size: 1, 2, 4, 8, 16, 32, 64, 128, and 256. Although 512 is technically a viable window, it cannot

slide across a 512 x 512 image so it is excluded. The successful isolation of LiDAR light points relies on the ability of the sliding window to encapsulate multiple LiDAR light points so that Otsu's Method thresholds the image based on the high frequency of their pixel intensities. Sliding windows of size 1 and 2 cannot realistically encapsulate multiple LiDAR light points. Windows of size 4 and 8 likely cannot either, but the possibility is considered.

Figure 12 shows the results of performing SWOM on the iPhone LiDAR light pattern depicted in Figure 8(a). Window sizes 4 and 8 fail to isolate the LiDAR light pattern, but window size 16 quickly improves on this. Window size 32 clearly displays the pattern while window sizes 64, 128, and 256 also clearly display the pattern. This illustrates the various window sizes' performance on LiDAR light in a dark environment, but smaller windows perform better on backgrounds with diverse lighting since they are more likely to encapsulate low variance lighting patterns which allows Otsu's Method to better isolate the LiDAR light points. Window sizes 32 and 64 both appear to accurately isolate the LiDAR light points, but testing is required to confirm which performs better.

However, note in Figure 13 how the 2D DFT of the result from Otsu's Method first presented in Figure 9 contains more noise than the 2D DFT of the result from SWOM with a window of size 64. The neural train trains to detect the LiDAR light pattern based on the 2D DFT image, so the more distinct result from SWOM improves the performance of the neural network.

3.2.6 Affine Transformation Property

As described in Section 2.4.9, affine transformations perform 3D transformations on a 2D plane. If cast upon a single flat surface, direct ToF LiDAR sensors similarly project a 2D plane of light points into 3D space. Therefore, affine transformations

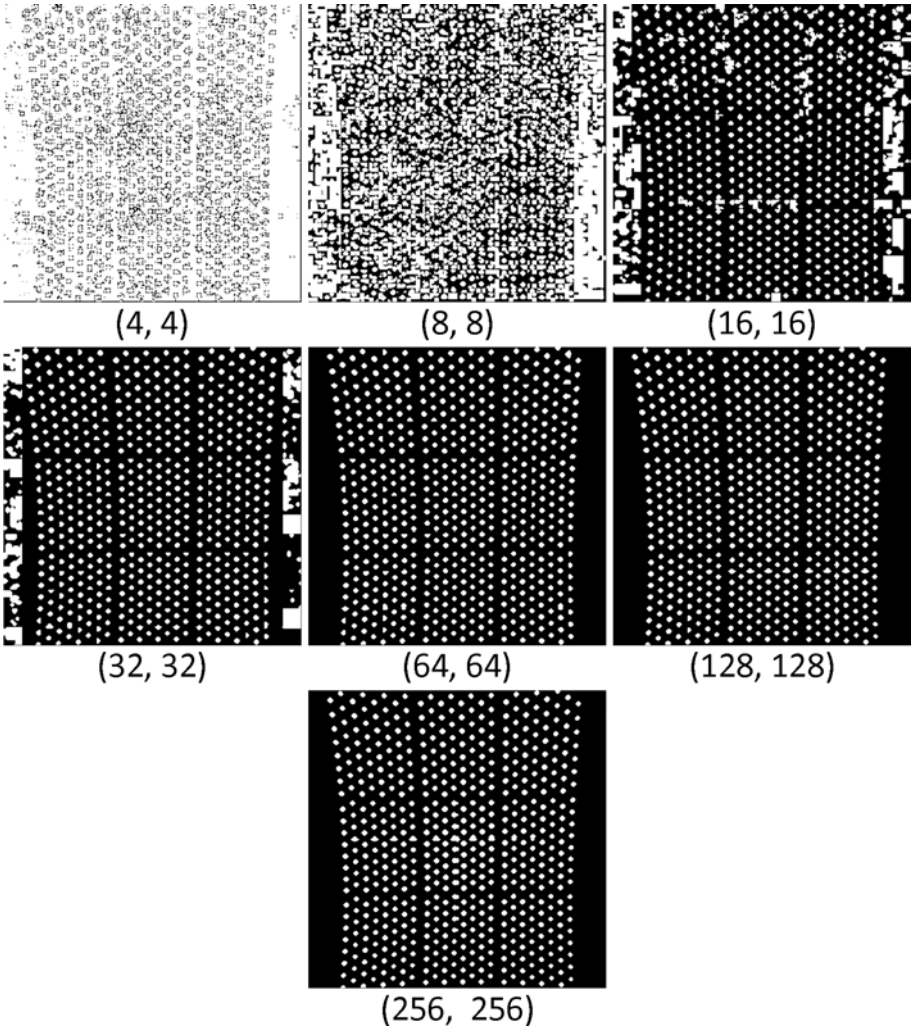

Figure 12: Sliding Window Otsu's Method window sizes

of the LiDAR light points simulate different positions of the phone in the real world. With a single image of the LiDAR light point pattern, these transformations produce an entire training dataset for the neural network. Training the neural network on the transformations as well as subsections of the transformations allows the neural network to classify LiDAR light points in images even if the security camera only captures a small portion of them.

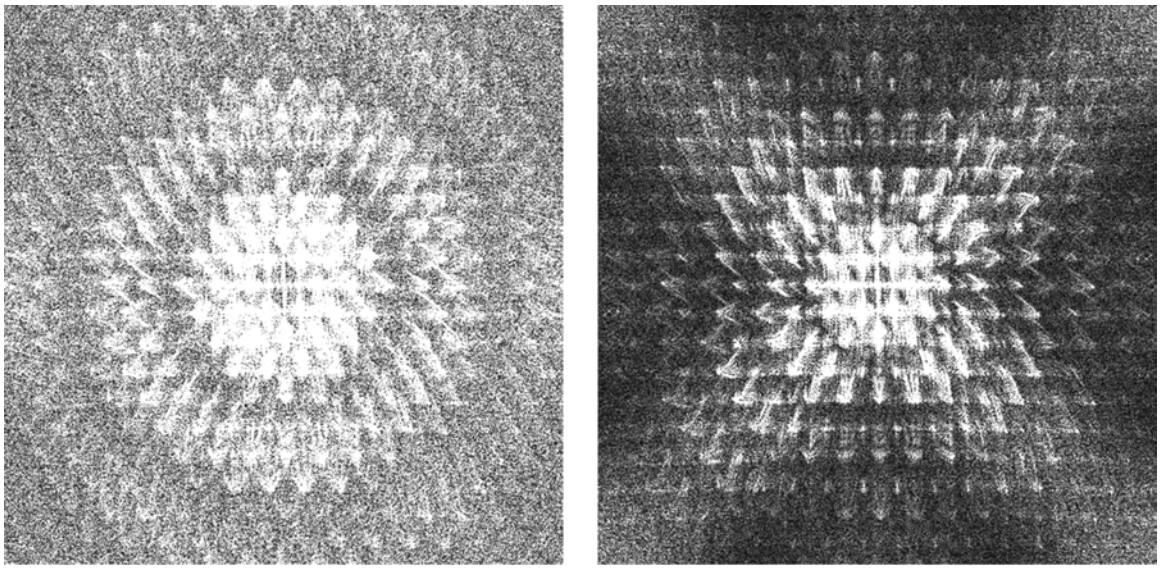

Figure 13: 2D DFT of Otsu's Method vs. SWOM

3.3 System Summary

These design considerations narrow down the possible designs of the system and justify the chosen system design. The following sections defines the system's three distinct components: the camera, the cellphone, and the Python scripts.

To prove the first hypothesis, a security camera configuration that records LiDAR light points in every frame is first identified. To prove the second hypothesis, a method is developed to process security camera footage into images to train the neural network and evaluate how accurately it identifies the presence of cellphone LiDAR light in images. Section 2.5.1 establishes that the neural network pattern classifier analyzes the frequency of pixel intensities at various locations in an image to decide its class. Section 3.2.5 also establishes that SWOM isolates the LiDAR light points which results in a binary image. Since pixels in a binary image can only have one of two intensity values, the classifier instead primarily analyzes the frequency of the pixel locations. Therefore, the neural network requires a training dataset with

many different permutations of where the LiDAR light points may appear in security camera footage. These permutations are generated with affine transformations of a single image of the cellphone LiDAR light pattern. Once the neural network is trained on these simulated images of cellphone LiDAR light, it is evaluated on how accurately it classifies real life images of cellphone LiDAR light captured by the security camera. The following sections summarize the designs of the security camera, cellphone, and Python script components that accomplish these tasks.

3.3.1 Camera Network Setup

As seen in Figure 14, the camera setup consists of a single security camera that connects to a desktop computer to which it exports videos. The camera is powered by Power over Ethernet (PoE), so it connects directly to a PoE port on a PoE switch. The security camera's instructions indicate that if the camera connects to a PoE switch instead of directly to its storage location, it must then connect to another network device before connecting to the storage device [30]. The workstation is the storage device in this situation, so the PoE switch connects to a second switch before it then connects to the workstation. Once connected, the camera advertises its preset Internet Protocol (IP) address to the workstation via the Address Resolution Protocol (ARP). Once the workstation receives the advertisement and records the IP address, it can connect to the camera with the NACL Web Plug-in to change the camera settings.

Figure 14: Camera connected to desktop computer. Red lines indicate Ethernet cable connections [30][93][94][95]

The switches illustrated in Figure 14 are abstracted in the box labelled **N** in Figure 15. This figure is an overhead depiction of the camera setup that shows the camera positioned 1 meter away from the wall towards which it points. It also sits 1.5 meters away from the left wall of the room. This room is illuminated by a 1000 Lumen LED in the lower left corner of the room. This light sits 1.73 meters above the ground, .2 meters away from both adjacent walls, and approximately 3.65 meters from the center of the wall portion captured in the security camera footage.

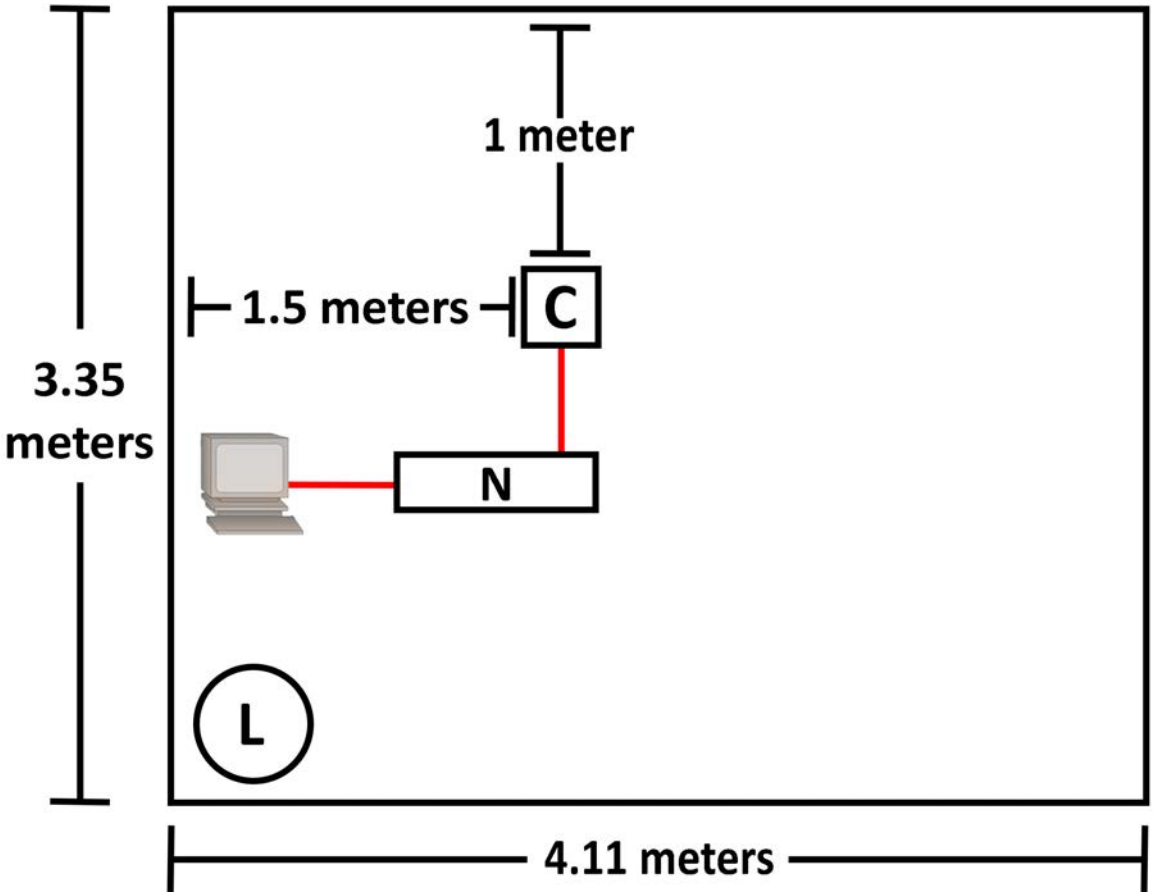

Figure 15: Overhead depiction of camera setup. (C) Security camera connects to (N) switches which connect to desktop [95]. (L) Light illuminates the environment. Red lines indicate Ethernet cable connections.

3.3.2 Cellphone Setup

The cellphone setup originally consisted of three cellphones, but the design considerations eliminate two phones leaving just the iPhone 12 Pro Max. The Xcode IDE loads the open source 3D scanner app onto the iPhone. The iPhone 12 Pro Max is positioned below and in line with the front of the camera such that the LiDAR sensor highlighted in Figure 16(a) sits as close to the camera sensor as possible. Both the cellphone and camera sit 1 meter off the ground and point perpendicularly to a flat wall 1 meter away as seen in Figure 16(b). This setup enables quick collection of video footage containing LiDAR light points.

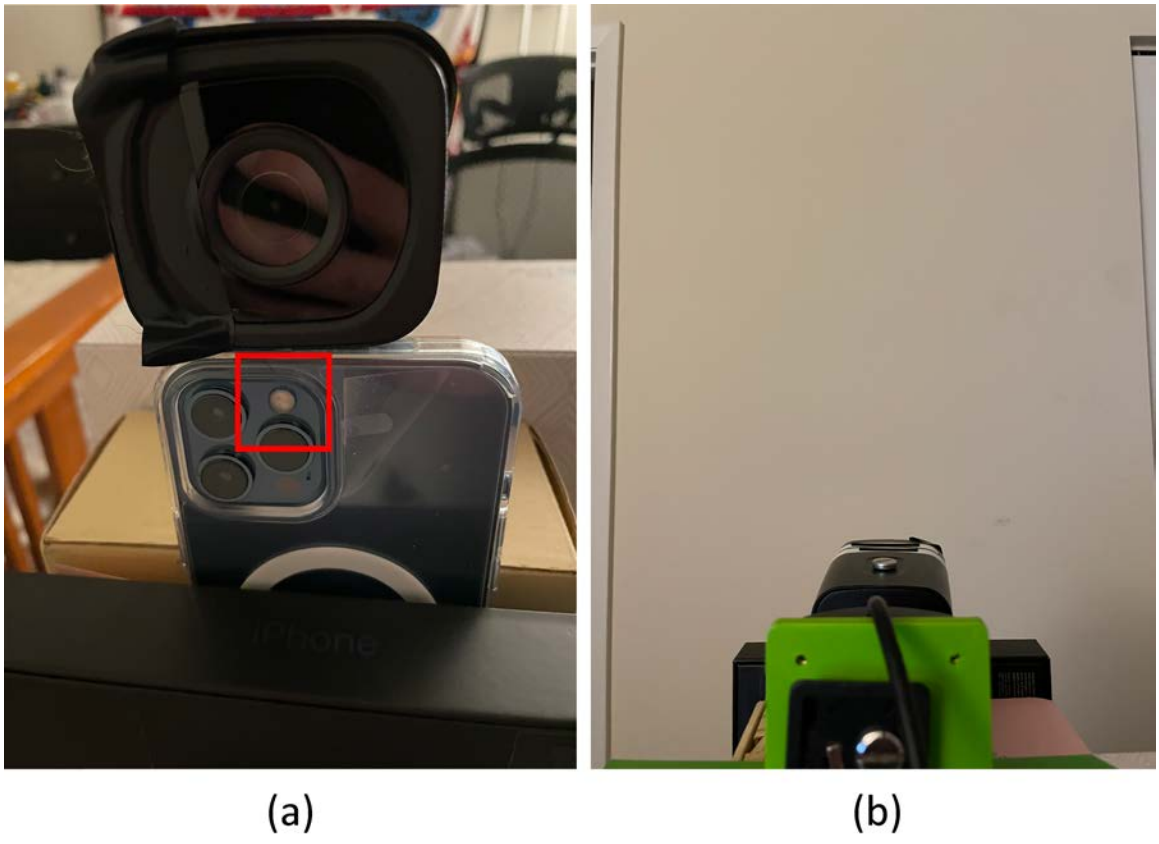

Figure 16: (a) Camera positioned above iPhone 12 Pro Max with LiDAR Sensor Highlighted and (b) rear view of camera pointed at wall 1 meter away

3.3.3 Python Scripts

The Python scripts perform three main tasks: recording video footage from the security camera and storing it on the workstation, creating the training and testing datasets, and training and testing the neural network. The inputs and outputs for each task are stored to disk, so the scripts for each task never directly interact with each other. Figure 17 illustrates the script pipeline.

The script that records the video footage, Capture_Footage.py, uses OpenCV to open a live feed to the video camera and record a certain length video to the workstation when a hotkey is pressed. The stream stays open and records a new video whenever the 'v' key on the workstation's keyboard is pressed. The script records new videos upon key press until the script stops running.

The script that processes the security camera footage in preparation for the neural network, Preprocess_Footage.py, implements the ideas explained in Section 3.2.5 and Section 3.2.6 to produce test data and training data. It takes a single photo of LiDAR light points and generates an entire training dataset using the following steps:

1. Isolate LiDAR light points with image thresholding
2. Extract square image of the LiDAR light points
3. Affine transform the image a given number of times
4. Slice each transformation into patches
5. Discard low quality patches
6. Rotate remaining patches a given number of times
7. Apply Sliding Window Otsu's Method
8. Save images to disk

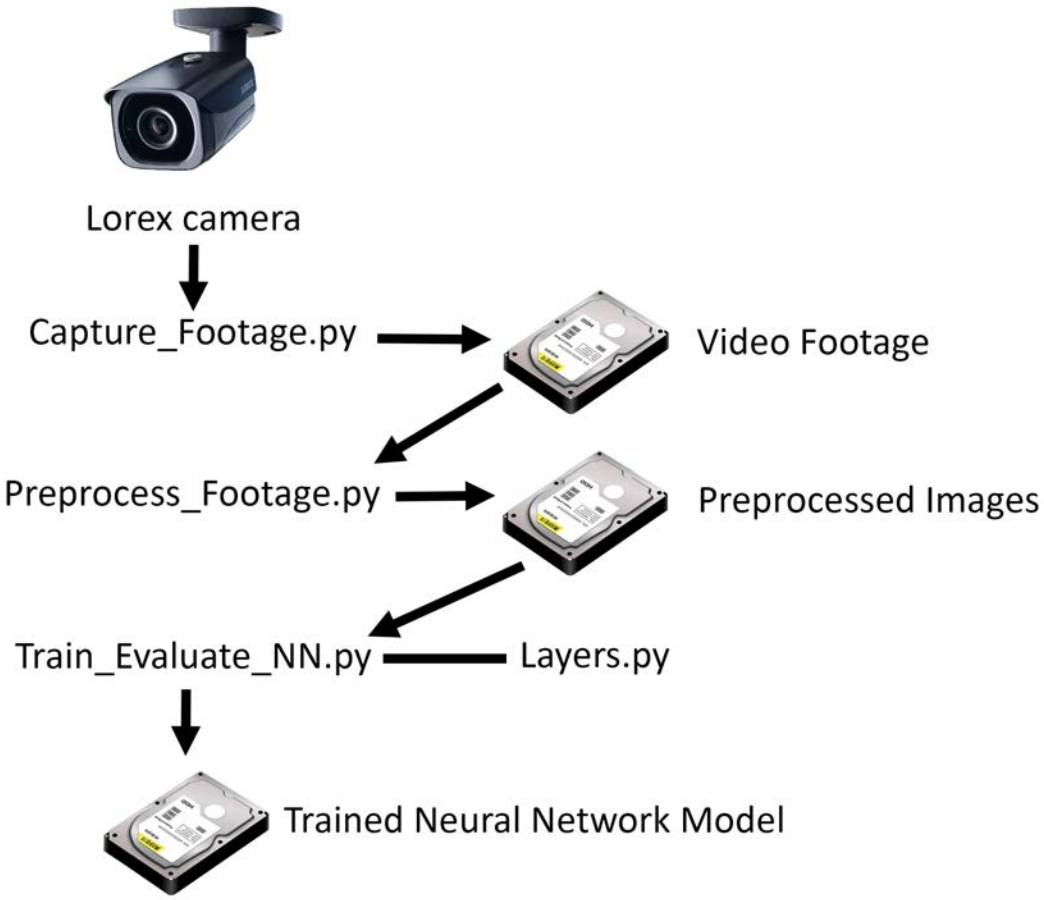

Figure 17: Python scripts pipeline [30][96]

The iPhone LiDAR light point pattern seen in Figure 3 needs to be standardized before it can be permuted. Steps 1 and 2 find the polygon that outlines all of the light points in the image then affine transforms it to a square. This is difficult to visually illustrate because the change is minimal, but this ensures that all the subsequent affine transformations in step 3 start with a standardized square image of the LiDAR light points. Step 3 receives a parameter containing the number of times to affine transform the image. Each affine transform is unique and is subsequently sliced into patches. Step 4 slices each transformation into 4 patches and slices each of those into 4 more patches for a total of 21 patches (1 original image + 4 patches + 16 patches) just like the neural network classifier does in Section 2.5.1. Step 5 discards any patches that do not contain more than 16 LiDAR light points or if the LiDAR light points do not cover the majority of the patch. This prevents the neural network from training on images that do not contain a significant amount of LiDAR light. Step 6 then receives a parameter containing the number of times to randomly rotate each of the remaining patches to increase the variety of the training dataset. Finally, step 7 applies SWOM to each patch rotation and saves it to the hard drive in step 8 for later use in training the neural network.

This script produces the test dataset using a subset of the steps used to create the training dataset. Rather than transforming one real image to simulate many real images, the test dataset consists exclusively of real images recorded from the security camera. Processing these images consists of the following steps:

1. Extract images from each frame of video
2. Apply Sliding Window Otsu's Method
3. Save images to disk

Since the test dataset only contains real images, no transformations except for

SWOM are needed. This script extracts a square image from the center of each frame in a video recorded by the security camera, resizes it to the proper size for the neural network, and applies SWOM. It saves the images containing LiDAR light to use as positive samples in the test dataset, and it saves the images that do not contain LiDAR light to use as negative samples in the test and training dataset. After this script creates the training and test datasets, the next script uses these datasets to train and evaluate the neural network.

The third script, Train_Evaluate_NN.py, creates the neural network, trains it on the training dataset, evaluates it on the test dataset, then saves the model along with its trained variable values to the hard drive. This script builds the model seen in Figure 18 from [11], but it replaces the leaky ReLU activation function with ReLU and the softmax output function with sigmoid. The leaky ReLU model performs well, but the three similar models that use ReLU in the paper perform better overall. Softmax can classify multiple classes, but only a sigmoid function is required to classify between two classes.

The model consists of four primary layers: input layer, processing layer, hidden layer, and output layer. The input layer formats images for the processing layer. The processing layer consists of three layers imported from the Layers.py script. The first layer splits the images into patches. The second layer calculates the 2D DFT magnitude of each patch and sums ("pools") the magnitude pixels for each ring. The third layer multiplies each ring sum by its respective ring weight which it passes to the hidden layer. The hidden layer applies the weights and biases from each ReLU neuron and hands the resulting values to the output layer consisting of a single sigmoid neuron that classifies the input image. The model uses a binary cross entropy loss function to feed back the classification results and update the trainable variables to improve classification accuracy. The trainable variables of this model are the ring

weights, ReLU weights, and ReLU biases.

This concludes the summary of the system design. The following section describes the implementation of the system in detail.

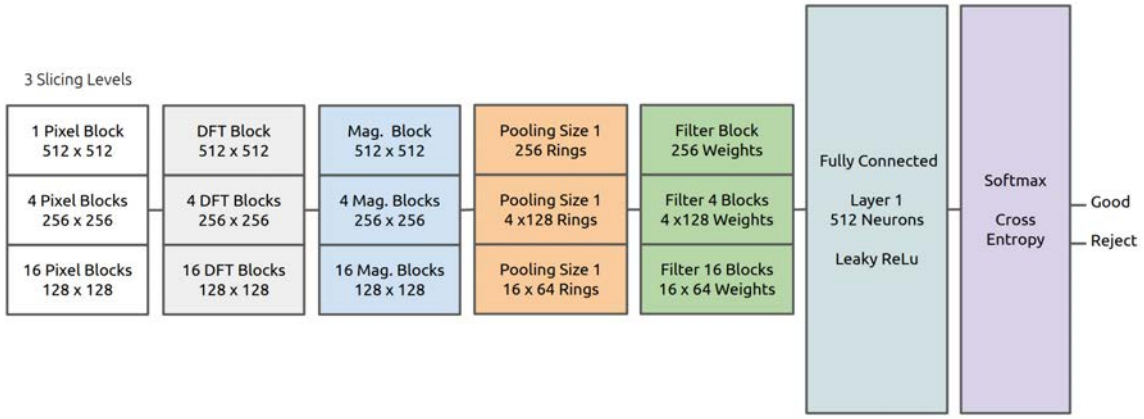

Figure 18: Sliced frequency layer architecture [11]

3.4 Detailed Description of System

This section details the implementation of the system in two sections: data collection and data processing. The data collection step describes how to set up the camera and capture LiDAR light from the cellphone. The data processing step explains the Python scripts that process the recorded videos, create a neural network, and train and evaluate the neural network.

3.4.1 Camera Configuration

The security camera connects to the workstation as described in Section 3.3.1. The NACL Web Plug-in on the workstation displays a search bar when opened. Typing the camera's IP address into this search bar presents a login page. Once logged in, a live stream from the camera shows what the camera sees. As labeled in Figure 19, click the "Setup" tab in box 1 then go to "Settings" under the "Camera" drop down menu in box 2. Box 3 shows the area that contains most of the camera settings.

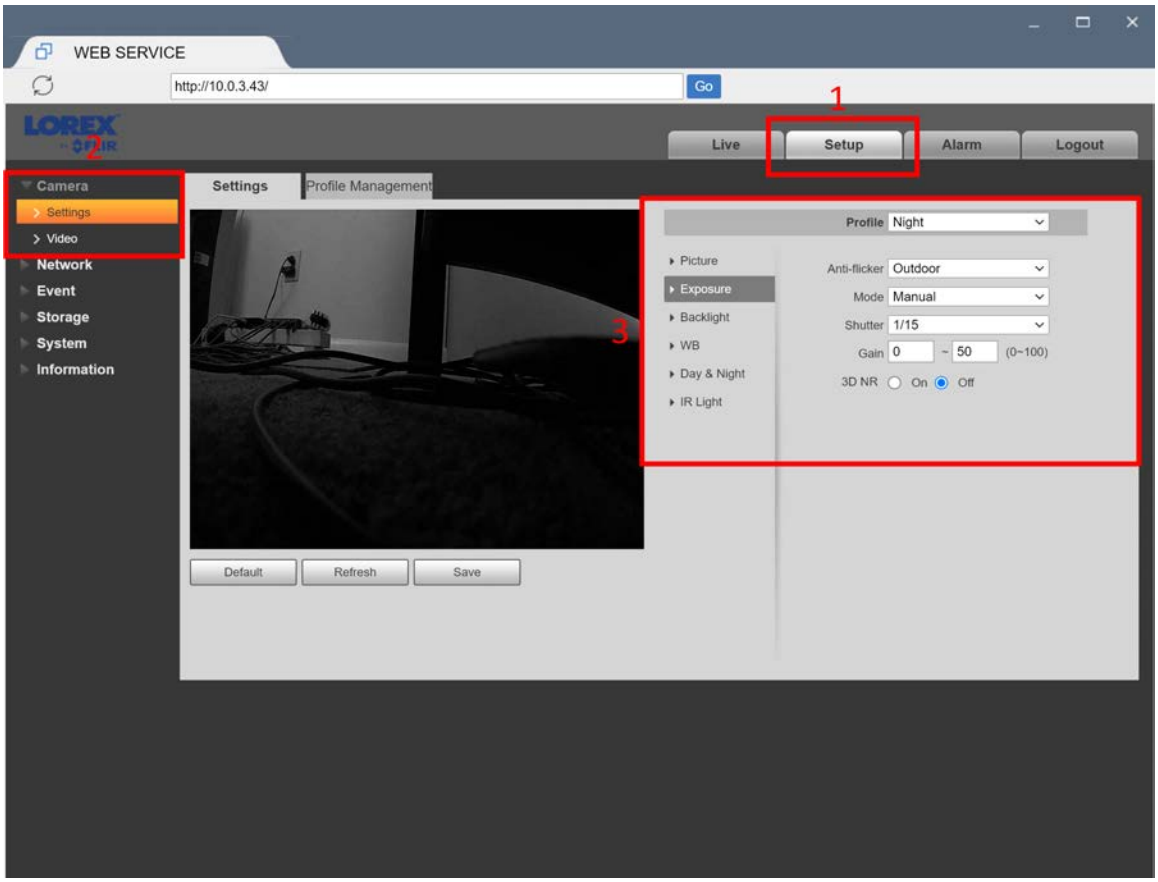

Figure 19: Camera settings in NACL Web Plug-in

The "Profile" drop down selects "Night" to disable the NIR filter. With night mode enabled, the next settings categories are on the left of box 3 starting with "Picture". Brightness, contrast, saturation, sharpness, and gamma remain at their default values of 50. Mirror and flip also remain at their default values of off and 0 degrees respectively. In the "Exposure" category, anti-flicker is set to outdoor, mode set to manual, and shutter set to 1/15. Leave gain at the default range of 0-50 dB and 3D NR off. In the "Backlight" category, leave mode off. In the "WB" category, set mode to outdoor. In the "Day & Night" category, set mode to B/W. In the "IR Light" category, leave mode off. Finally, click "Save" underneath the preview of the video stream to save these settings. These settings are summarized in Table 2.

Next, click the "Video" tab highlighted by box 1 in Figure 20 to edit the settings

Table 2: Security camera settings

Category	**Setting**	**Value**
Picture	Brightness	50
	Contrast	50
	Saturation	50
	Sharpness	50
	Gamma	50
	Mirror	Off
	Flip	0 degrees
Exposure	Anti-flicker	Outdoor
	Mode	Manual
	Shutter	1/15
	Gain	0-50
	3D NR	Off
Backlight	Mode	Off
WB	Mode	Outdoor
Day & Light	Mode	B/W
IR Light	Mode	Off

of saved videos. Box 2 highlights the most notables settings. Leave "Encode Mode" on H.264H and "Super H.265/H.264" off. Set resolution to 1080P and FPS to 15. Keep "Bit Rate Type" on CBR, but set "Bit Rate" to the maximum, non-custom value of 6144. The bit rate can be set to a higher custom rate, but a non-custom value better represents the typical security camera settings others may use.

The camera is now completely configured and prepared to record video footage. A Python script records the video footage, so the NACL Web Plug-in is no longer needed after configuring the camera settings. To prepare to record footage, position the camera 1 meter off the ground and 1 meter from a flat wall at which it points perpendicularly. The flat wall is covered in off-white house paint which reflects LiDAR light fairly well.

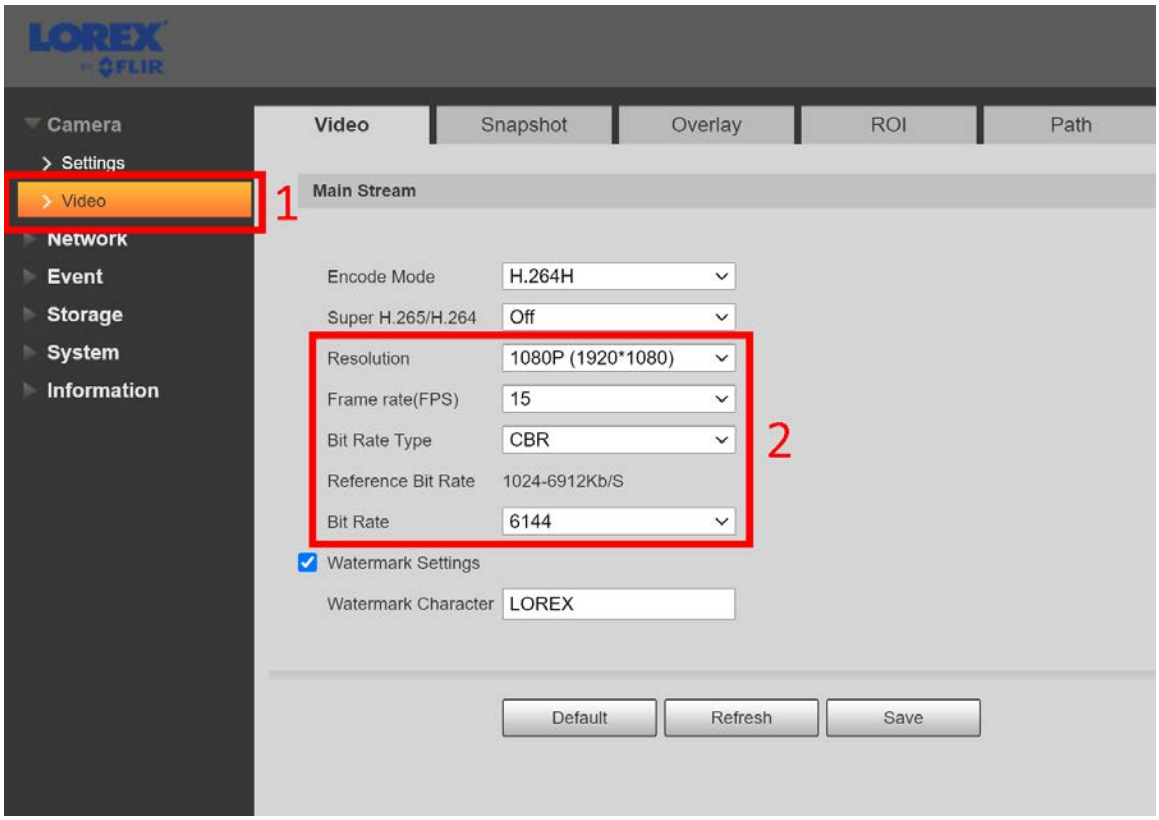

Figure 20: Video settings in NACL Web Plug-in

3.4.2 Video Capture Python Script

The Capture_Footage.py Python script is adapted from an online source to record video footage from the security camera [97]. Items in the following listings surrounded by arrows <like this> indicate redacted information that needs to be filled in for the script to work. This script connects to the camera, launches a live stream, and records a video of certain length each time a hotkey is pressed.

```
1 import time
2 import datetime
3 import os
4 from threading import Thread
5 import cv2
6
7 dirOut = <path>
8 fps = 15
9 cameraIP = <IP address>
10 video_length = 30
11 user = <username>
12 pwd = <password>
```

Listing III.1: Beginning of Capture_Footage.py

Listing III.1 displays the beginning of the Capture_Footage.py script. Lines 1 - 5 import the necessary libraries for this script to use. Line 7 stores the file path of where to save the videos on the hard drive. Line 8 sets the FPS to 15 and the variable **cameraIP** stores the camera's IP address, **video_length** defines the length of the recorded video in seconds, and **user** and **pwd** store the username and password credentials respectively to authenticate to the camera.

```
1 if __name__ == '__main__':
2     # Connects with High Resolution 1920x1080
3     src = 'rtsp://' + user + ":" + pwd + "@" + cameraIP + ":554/cam/
    realmonitor?channel=1&subtype=0"
4     video_stream_widget = RTSPVideoWriterObject(src)
5     while True:
6         try:
7             video_stream_widget.show_frame()
8         except AttributeError:
9             pass
```

Listing III.2: Capture_Footage.py main function

The main function of Capture_Footage.py seen in Listing III.2 concatenates the **user**, **pwd**, and **cameraIP** variables into a Uniform Resource Locator (URL) on line 3 used to connect to the camera. It passes the URL to a custom RTSPVideoWriter-Object class constructor on line 4 which returns a streaming object that displays a live stream to the camera on line 7. The live stream waits for the press of a hotkey to record a video.

```
1 class RTSPVideoWriterObject(object):
2     def __init__(self, src):
3         # Create a VideoCapture object
4         self.capture = cv2.VideoCapture(src)
5 
6         # Default resolutions of the frame are obtained (system
    dependent)
7         self.frame_width = int(self.capture.get(3))
8         self.frame_height = int(self.capture.get(4))
9         self.last_cap = time.time()
10        self.write = 0
```

```
11
12          # Start the thread to read frames from the video stream
13          self.thread = Thread(target=self.update, args=())
14          self.thread.daemon = True
15          self.thread.start()
```

Listing III.3: RTSPVideoWriterObject class

The RTSPVideoWriterObject class constructor seen in line 4 of Listing III.3 uses OpenCV (cv2) with the URL to create a live video capture connection to the camera. Lines 7-10 store the width and height of the video stream, the time it starts the live stream, and a **write** boolean to know if a stored frame is available to write to a video on the hard drive. Lines 13-15 then launch the update function in a separate thread to read images from the video stream in the background.

```
1   def update(self):
2       # Read the next frame from the stream in a different thread
3       while True:
4           if self.capture.isOpened():
5               if(time.time() - self.last_cap >= 1/fps):
6                   self.last_cap = time.time()
7                   self.write = 1
8                   (self.status, self.frame) = self.capture.read()
```

Listing III.4: update() function

The update function seen in Listing III.4 runs in the background in an infinite loop to capture images from the camera. Line 4 checks if it is connected to the camera. Based on the set FPS, it checks in line 5 if enough time has elapsed since it last saved an image. If enough time has elapsed, lines 6-8 update the time at which it last saved an image, ensure the write variable indicates that images are available to save to disk, then read an image from the camera stream. After this update function

launches in the background, the class constructor returns a class instance which executes the show_frame() function seen in line 7 of Listing III.2.

```
1     def show_frame(self):
2         if self.status:
3             cv2.imshow('frame', self.frame)
4         key = cv2.waitKey(1)
5         if key == ord('v'):
6             self.save_vid()
7         elif key == ord('q'):
8             self.capture.release()
9             self.output_video.release()
10            cv2.destroyAllWindows()
11            exit(1)
```

Listing III.5: show_frame() function

In Listing III.5, line 3 of the show_frame() function displays a video frame if the status flag on line 2 indicates a frame is available to show. If no key is pressed, this function keeps displaying frames when they are available which results in a live stream of the connected camera at 15 FPS. Line 5 recognizes when the 'v' key is pressed and calls the save_vid() function on line 6. Line 7 recognizes when the 'q' key is pressed, destroys the video stream, and exits.

```
1     def save_vid(self):
2         filename = f"{datetime.datetime.now():%Y-%m-%dT%H%M%S}" + '.avi'
3         file = os.path.join(dirOut, filename)
4         codec = cv2.VideoWriter_fourcc('M', 'J', 'P', 'G')
5         self.output_video = cv2.VideoWriter(file, codec, fps, (self.
      frame_width, self.frame_height))
6         for i in range(video_length * fps):
```

```
7               while True:
8                   if self.write:
9                       self.output_video.write(self.frame)
10                      self.write = 0
11                      break
12          self.output_video.release()
```

Listing III.6: save_vid() function

Finally, pressing the 'v' key executes the save_video() function in Listing III.6. Line 2 creates a filename for the video based on the current date and time which line 3 appends to the directory defined in Listing III.1. Line 4 sets the proper video codec that aligns with the H.264H Encode Mode set in the camera's video settings. With all these in place, line 5 creates a file handle to the output video and lines 6-11 write frames to the video. Line 6 ensures that the number of frames written to the video file corresponds to the set video length and FPS. Line 7 waits until a frame is available to write. Once line 8 indicates a frame is available, line 9 writes it to the output video file and line 10 resets the **write** boolean to indicate that there is no longer a frame available to write to the video. After the for loop on line 6 writes every frame, line 12 deletes the handle to the output video file.

When the save_video() function returns, the script continues to call the show_frame() function in a loop. Each time the 'v' key is pressed, it calls the save_video() function and saves another video to disk. This loop continues until the 'q' key is pressed to exit the script. This concludes the explanation of Capture_Footage.py which is included in Appendix A for reference. With the camera configured and script set up, the security camera can record cellphone LiDAR light.

3.4.3 Capture Cellphone LiDAR Light

The LiDAR light pattern displayed in both Figure 3 and Figure 12 possesses diagonal symmetry. This means that a 180 degree rotation of LiDAR light points projected from the left of the camera simulates LiDAR light points projected from the right of the camera.

Figure 21 shows cellphone LiDAR projected onto the wall from the left of the camera in a slightly downward direction. Rotating this image 180 degrees simulates LiDAR projection from the right of the camera in a slightly upward direction as shown in Figure 22. This symmetry saves time in production of both the training dataset and the test dataset.

This correlates the ability to detect a LiDAR pattern with the ability to detect the same LiDAR pattern rotated 180 degrees. Therefore, the test dataset consists of real LiDAR light images projected from only half of the rotational positions as illustrated in Figure 23. This image points perpendicularly at the wall from behind the camera, so the box labelled "C" represents the rear of the camera. Each box labelled "P" represents a location at which the iPhone is held. At each location, the phone points to the center of the wall where the camera points. The phone is first held adjacent to the camera, moved to the top right position, moved to the top middle then top left position, moved to the middle left then bottom left position, then finally moved back to middle position adjacent to the camera. Described more simply, the phone rotates amongst the positions counterclockwise starting and ending adjacent to the camera.

To capture LiDAR light projected at each position, the 3D scanning app is launched on the iPhone which is then positioned adjacent to the camera. After running the Capture_Video.py script and pressing the 'v' key to record a video, the phone is moved to each position across a span of 30 seconds while continuously pointing towards the middle of the wall. The phone remains at each position for approximately

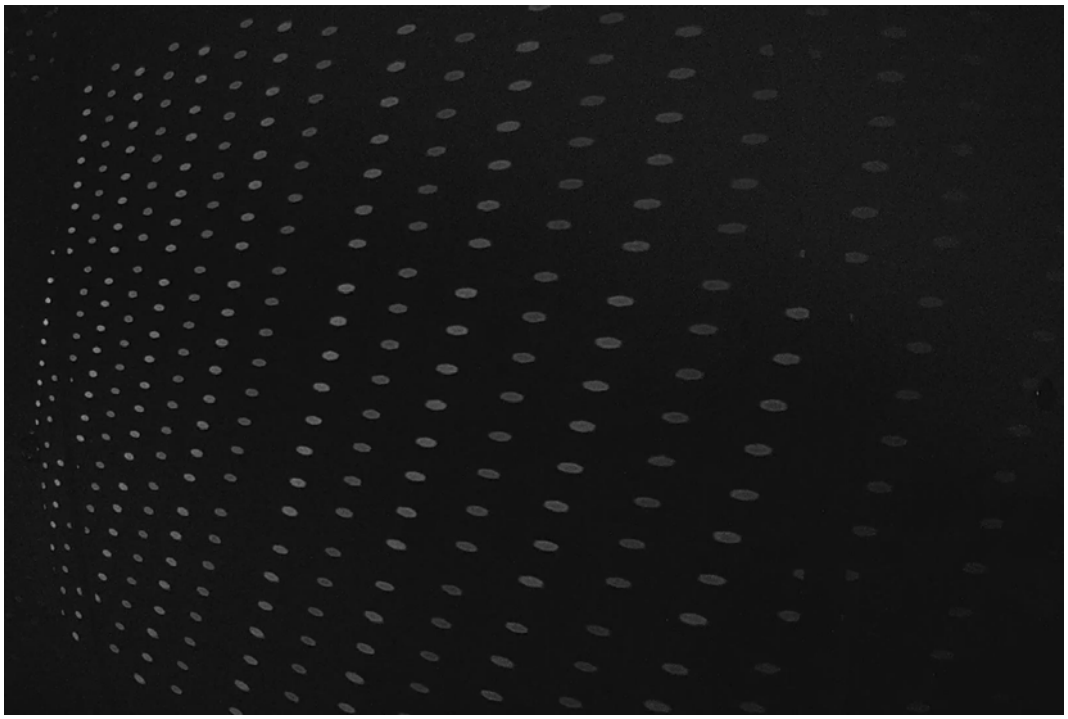

Figure 21: Projection of LiDAR from left of camera

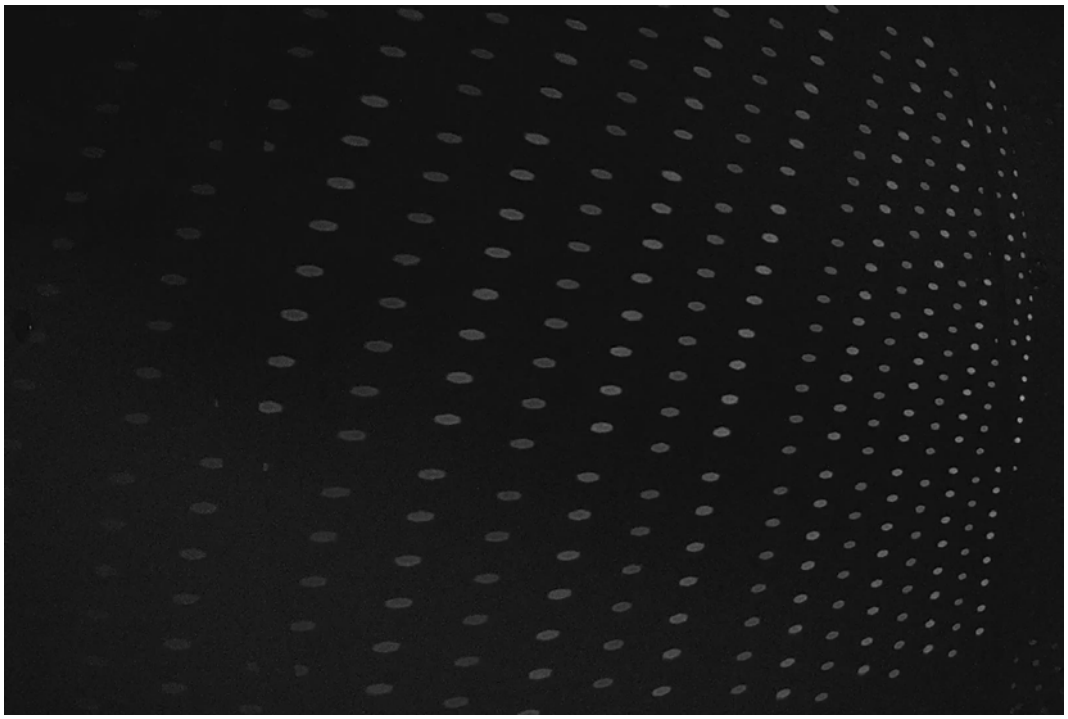

Figure 22: Simulated projection of LiDAR from right of camera

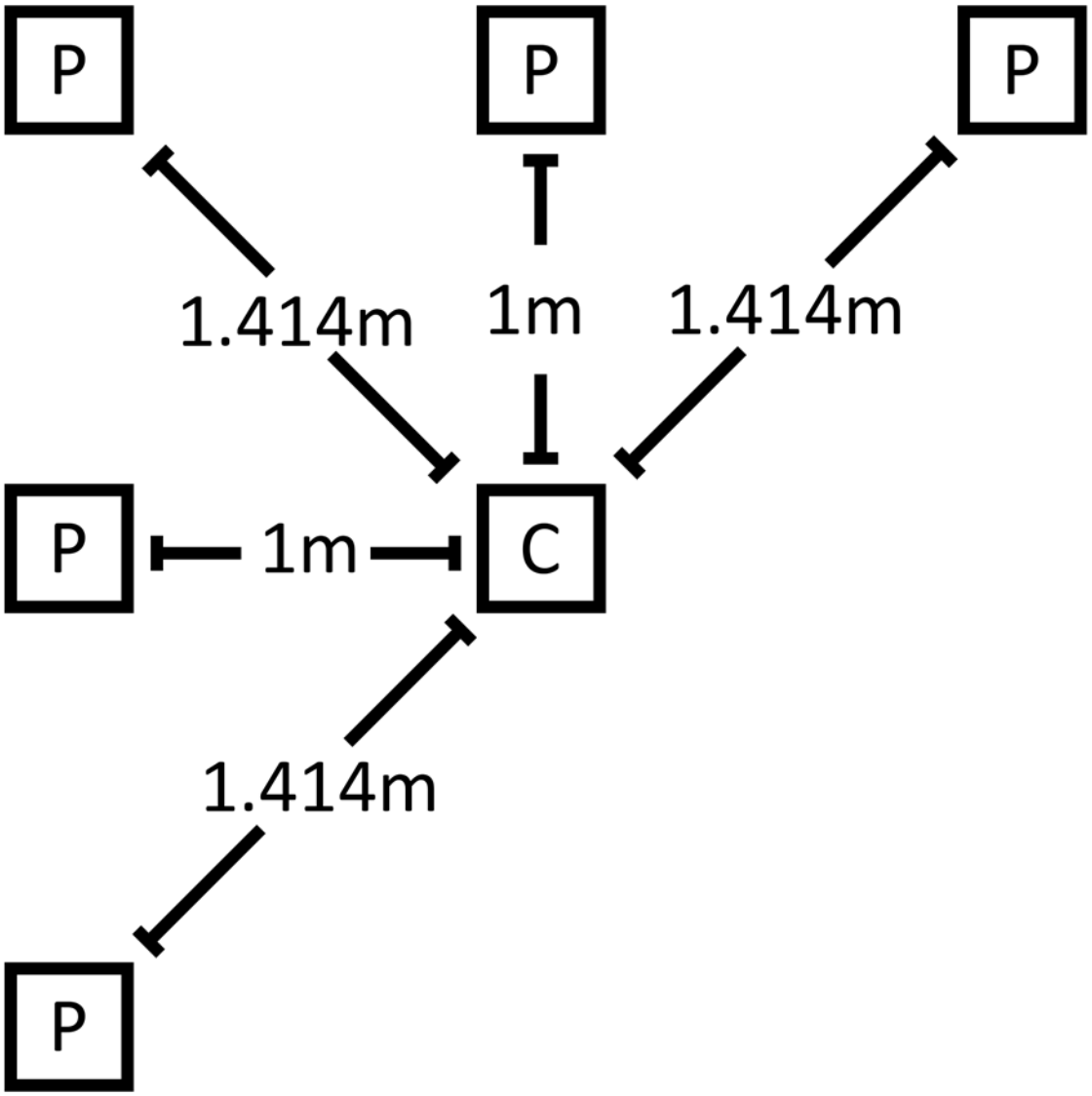

Figure 23: Positions of iPhone during LiDAR projection

4 seconds before moving to the next position. The movement is all done by hand to evaluate how well a neural network identifies LiDAR light subject to aberrations caused by human movement. After capturing a 30 second video with LiDAR light in every frame, the 3D scanning app is closed and the 'v' key is pressed to record a 30 second video with no LiDAR light in any frame. These two 30 second videos comprise all the data needed to produce a test dataset for a single light level, but footage is captured in a room with an LED light turned on and off. The Photon light meter app measures 35 Lux with the light on and 0 Lux with the light off. Therefore, a video with and without LiDAR light are captured in the 35 Lux environment and a video with and without LiDAR light are captured in the 0 Lux environment.

This concludes the data collection portion of this research. The following sections detail how the collected video footage is processed and used to train the neural network.

3.4.4 Processing Python Script

The processing Python script, Preprocess_Footage.py, generates the training dataset from a single image of LiDAR light points and extracts the test dataset from the videos captured in Section 3.4.3. Appendix B contains the entire script for further reference. This section first explains the functions that extract the test dataset images from the recorded videos followed by an explanation of the functions that generate the training dataset.

A path must be provided to the dataset creation function. This path locates a folder on the hard drive with the following internal structure:

- path\
- path\test\

- path\test\videos\

The **path\test\videos** folder contains videos from Section 3.4.3 to process into the test dataset. At minimum, the videos folder contains two videos captured in a 0 Lux environment: one video with LiDAR light in each frame called **vid_name.avi** and one video with no LiDAR light in any frames called **vid_name_neg.avi**. To produce a test and training dataset for a 35 Lux environment, an identical but separate file structure contains the videos with the LED light turned on. This research labels the 0 Lux path as **path** and the 35 Lux path as **path_light**. The **videos** folder may contain more videos, but they must be in pairs of one with LiDAR light in each frame and one with no LiDAR light in any frames. The videos without LiDAR light must include **_neg** in the video name.

After executing this script to produce the test and training datasets, the path directory takes on the following internal structure:

- path\
- path\model\
- path\test\
- path\test\masked_pics\
- path\test\raw_pics\
- path\test\videos\
- path\train\
- path\train\lidar\
- path\train\neg\

The **path** folder now contains three main folders: model, test, and train. TensorFlow saves and loads the neural network to the **path\model** folder. TensorFlow abstracts this process behind simple Python commands, so the exact format of this folder does not concern this research. The **path\test** folder now contains a **raw_pics** folder of all the images extracted from the videos as well as a **masked_pics** folder containing the result of applying SWOM to every picture in **raw_pics**. The **path\train** folder contains a **lidar** folder of simulated LiDAR light point images and a **neg** folder of images with no LiDAR light points.

When the TensorFlow dataset generator looks in a folder, it generates images classes based on the names and contents of the folders. Therefore, the dataset generator produces a dataset from **path\train** with two classes: lidar and neg. The lidar class contains images with LiDAR light from the **path\train\lidar** folder, and the neg class contains images without LiDAR light from the **path\train\neg** folder. This dataset generator also produces a test dataset for each folder in **path\test\masked_pics**, so each of the folders in **masked_pics** must similarly contain a **lidar** and **neg** folder. For each pair of videos **video_name.avi** and **video_name_neg.avi** that do and do not contain LiDAR light respectively, their final processed images are stored in **path\test\masked_pics\video_name\lidar** and **path\test\masked_pics\video_name\neg** respectively. This nested folder structure is slightly complicated, but it simplifies dataset generation when training and testing the neural network.

For use in explaining this script, the term 'LiDAR images' refers to the collection of images that contain LiDAR light whether used in a test or training dataset. Similarly, the term 'neg images' refers to the collection of images that do not contain LiDAR light whether used in a test or training dataset. The following listings explore how the processing script produces both datasets.

```
1 if __name__ == '__main__':
2     start_time = time.time()
3     create_test_and_train_data('path', win_size=64, num_transforms=16,
    num_rotations=3, img_size=512)
4     print("%f seconds" % (time.time() - start_time))
```

Listing III.7: main() function

Listing III.7 shows the **main()** function of the Preprocess_Footage.py script. The **main()** function records the start time in line 2 which it uses in line 4 to calculate the run time of the **create_test_and_train_data()** function called on line 3. It calls this function with the argument 'path' for a 0 Lux environment and 'path_light' for a 35 Lux environment. As the name implies, this function creates the test and training datasets from the data contained in the path passed in the first argument. The **win_size** argument controls the size of the SWOM window. The **num_transforms** argument controls how many affine transformations are used to produce the training dataset. The **num_rotations** argument defines how many times each affine transformation is rotated to produce the training dataset. Finally, the **img_size** informs subsequent functions of the image width.

```
1 def create_test_and_train_data(path, win_size, num_transforms,
    num_rotations, img_size):
2     create_test_dataset(path, win_size=win_size, img_size=img_size)
3     create_train_dataset(path, num_transforms=num_transforms,
    num_rotations=num_rotations, win_size=win_size, img_size=img_size)
```

Listing III.8: create_test_and_train_data() function

Listing III.8 shows the **create_test_and_train_data()** wrapper function that controls the creation of both the test and training datasets. The training dataset uses neg images from the test dataset, so this function first passes its arguments to **cre-**

ate_test_dataset() on line 2 to create the test dataset then calls **create_train_dataset()** on line 3 to create the training dataset. Similar to how the **main()** function calls this function, other Python scripts can import and call this function with the proper arguments to create datasets.

```
1 def create_test_dataset(dataset_path, win_size=64, img_size=512):
2     test_data_path = dataset_path + "\\test"
3     if not os.path.isdir(test_data_path + '\\raw_pics'):
4         all_vids_to_frames(test_data_path, img_size)
5     convert_raw_pics_to_masked_pics(test_data_path, win_size)
```

Listing III.9: create_test_dataset() function

In Listing III.9, the received **dataset_path** argument contains the root **path** folder, so this function adds 'test' on line 2 to target the **path\test** dataset folder. If line 3 finds that the **raw_pics** folder does not already exist in this test dataset folder, line 4 calls the **all_vids_to_frames()** function to save every frame from the videos in **path\test\videos** to images in the **path\test\raw_pics** folder. The **convert_raw_pics_to_masked_pics()** function is then called on line 5 to apply SWOM to every image saved in **path\test\raw_pics** and save the results to **path\test\masked_pics**. The images in the **path\test\masked_pics** folder are the test dataset.

```
1 def all_vids_to_frames(path, img_size):
2     vid_path = path + '\\videos'
3     videos = []
4     for file in os.listdir(vid_path):
```

```
5           if '.avi' in file:
6               videos.append(os.path.join(vid_path, file))
7       for video in videos:
8           vid_to_frames(video, img_size)
```

Listing III.10: all_vids_to_frames() function

The **img_size** argument passed to **all_vids_to_frames()** on line 1 of Listing III.10 controls the size of the images saved to the test dataset. This argument is passed forward through multiple functions until it is needed to write the images to disk. The **path** argument passed to this function refers to **path\test**. On line 2, this function appends 'videos' to the path resulting in **path\test\videos**. Lines 3-6 find and record the full path of every video stored in the videos folder. Lines 7 and 8 then iterate through the path list and pass each full video path as well as the desired image size to the **vid_to_frames()** function on line 8. The **vid_to_frames()** function immediately calls the **create_pic_directories()** functions, so the next listing describes that function first.

```
1 def create_pic_directories(video_path):
2     split_path = video_path.split('\\')
3     vid_name = split_path[-1].split('.')[0]
4     parent_path = ('\\').join(split_path[:-2])
5     if '_neg' in vid_name:
6         vid_name = vid_name.replace('_neg', '')
7     raw_pics_path = parent_path + '\\raw_pics\\' + vid_name + '\\'
8     masked_pics_path = parent_path + '\\masked_pics\\' + vid_name + '\\'
```

Listing III.11: create_pic_directories() function

As seen in Listing III.11, this **create_pic_directories()** function creates the complete file structure to store all the raw images from a single test video. The

video_path argument passed to this function contains the full path to a single video in the format **path\test\videos\vid_name.avi** if the video contains LiDAR light or **path\test\videos\vid_name_neg.avi** if the video does not contain LiDAR light. From this full path, lines 2-4 extract the video name and the parent path **path\test**. Lines 5 and 6 remove the '_neg' string from the negative video file names so that videos with and without LiDAR light result in the same **vid_name**. This ensures that the LiDAR and neg images get stored to the same video folder in the **raw_pics** and **masked_pics** folders. Lines 7 and 8 then create the full path string for both of these folders.

```
1    # path\test\raw_pics\<vid_name>\lidar\
2    os.makedirs(raw_pics_path + 'lidar', exist_ok=True)
3    # path\test\raw_pics\<vid_name>\neg\
4    os.makedirs(raw_pics_path + 'neg', exist_ok=True)
```

Listing III.12: create_pic_directories() function

Once the path strings are created, the folders can be created. Listing III.12 shows how the **create_pic_directories()** function recursively creates the **raw_pics** folder structure on lines 2 and 4. Within the **raw_pics** folder, the <**vid_name**> folder stores all the images from **vid_name.avi** in **lidar** and all the images from **vid_name_neg.avi** in **neg**.

```
1    # path\test\masked_pics\<vid_name>\lidar
2    os.makedirs(masked_pics_path + 'lidar', exist_ok=True)
3    # path\test\masked_pics\<vid_name>\neg
4    os.makedirs(masked_pics_path + 'neg', exist_ok=True)
```

Listing III.13: create_pic_directories() function

Similarly to the **raw_pics** folders, Listing III.13 shows how the **create_pic_directories()** function recursively creates the **masked_pics** folder structure on lines 2 and 4. Within the **masked_pics** folder structure, the <**vid_name**> folder stores the results of SWOM applied to the images in **path\test\raw_pics**<**vid_name**>****. The SWOM results are stored in the corresponding <**vid_name**>**\lidar** and <**vid_name**>**\neg** folders in **path\test\masked_pics**. After creating these directories, this function returns back to the **vid_to_frames()** function that calls it.

```
1 def vid_to_frames(video_path, img_size):
2     raw_pics_path = create_pic_directories(video_path)
3     lidar_path = raw_pics_path + 'lidar'
4     neg_path = raw_pics_path + 'neg'
```

Listing III.14: vid_to_frames() function

The first half of **vid_to_frames()** can be seen in Listing III.14. This function immediately calls **create_pic_directories()** on line 2 so it has a place to store the frames of the video. Line 3 stores the path **path\test\raw_pics**<**vid_name**>**\lidar**. Line 4 stores the path **path\test\raw_pics**<**vid_name**>**\neg**. This function saves LiDAR images to the **lidar** folder and neg frames to the **neg** folder.

```
1     split_path = video_path.split('\\')
2     vid_name = split_path[-1].split('.')[0]
3
4     img_count = 0
5     cap = cv2.VideoCapture(video_path)
6     ret, og_img = cap.read()
```

Listing III.15: vid_to_frames() function

The next part of **vid_to_frames()** seen in Listing III.15 gets the **vid_name** on

lines 1 and 2 to target the correct folder in **raw_pics**. Line 4 initialises the counter **img_count** to label saved images. Line 5 opens a file handle to the video passed to this function via the **video_path** argument and reads the first frame into **og_img** on line 6.

```
1     root = video_path.split('\\')[0]
2     pattern_path = root + '\\light_pattern.png'
3         if not os.path.isfile(pattern_path) and 'neg' not in vid_name:
4         cv2.imwrite(pattern_path, og_img)
```

Listing III.16: vid_to_frames() function

Lines 1-4 in Listing III.16 then check if **path\light_pattern.png** exists. This is the file used to create the training dataset. If it does not exist, it saves the first frame of the video as that image. It also checks that 'neg' is not in the video name to ensure the saved light pattern image comes from a video with LiDAR light in it.

```
1     while og_img is not None:
2         frame_path = os.path.join(lidar_path, vid_name + "_" + str(
    img_count) + ".png")
3         if 'neg' in vid_name:
4             frame_path = os.path.join(neg_path, vid_name + "_" + str(
    img_count) + ".png")
```

Listing III.17: vid_to_frames() function

After creating **path\light_pattern.png** if it does not exist, this function finally loops through each frame of a video and saves it as seen in Listing III.17. It enters the reading loop in line 1 and formats the path at which to store the first image on line 2. The **frame_path** variable stores the path **path\test\raw_pics\<vid_name>\lidar\<vid_name>_<img_count>.png** to store LiDAR images. It alternatively

stores the path **path\test\raw_pics\<vid_name>\neg\<vid_name> _<img_count>.png** on lines 2 and 3 to store neg images.

```
1         crop = og_img[65:1015, 500:1450]
2         res = cv2.resize(crop, (img_size, img_size))
3         cv2.imwrite(frame_path, res)
4         img_count += 1
5         ret, og_img = cap.read()
```

Listing III.18: vid_to_frames() function

After forming the correct correct file path, the while loop continues in Listing III.18. Line 1 in the body of the loop extracts a 950 x 950 pixel square from the center of the video frame. Based on the test environment, this 950 pixel square is the maximum square size that does not overlap with door frames or other edges that may affect the SWOM result in the final image. However, the size of the extracted square can adapt to fit the testing environment. Line 2 then resizes this square to the **img_size** provided as an argument to this function. This research uses the image size 512 x 512 in accordance with the model described in Section 3.3.3. After resizing the image, line 3 finally saves the image to the appropriate **lidar** or **neg** folder in the respective **raw_pics\<vid_name>** folder. Given the video **path\test\vidoes\dark.avi**, this function saves the first frame to **path\test\raw_pics\dark\lidar\dark_0.png**.

Once this while loop reads, extracts, resizes, and saves an image from each frame of a video, it returns to **all_vids_to_frames()**. Once the loop in **all_vids_to_frames()** executes **vid_to_frames()** on every **video_path**, it returns to **create_test_dataset()**. At this point, all of the test videos are stored to the hard drive as images, so the processing script executes **convert_raw_pics_to_masked_pics()** to apply SWOM to the saved images.

```
1 def convert_raw_pics_to_masked_pics(dataset_path, win_size):
2     raw_pic_paths = []
3     for subdir, dirs, files in os.walk(dataset_path + '\\raw_pics'):
4         for file in files:
5             raw_pic_paths.append(os.path.join(subdir, file))
```

Listing III.19: convert_raw_pics_to_masked_pics() function

The **convert_raw_pics_to_masked_pics()** function finds the path to each raw image and applies SWOM to it. As seen in lines 1-4 of Listing III.19, it first stores the full path to every image saved in **path\test\raw_pics**.

```
1     for raw_pic_path in raw_pic_paths:
2         raw_pic = cv2.imread(raw_pic_path, cv2.IMREAD_GRAYSCALE)
3         masked_img = sliding_window_mask(raw_pic, win_size)
4         masked_pic_path = raw_pic_path.replace('raw_pics', 'masked_pics'
    )
5         cv2.imwrite(masked_pic_path, masked_img)
```

Listing III.20: convert_raw_pics_to_masked_pics() function

Listing III.20 then shows how the rest of the function applies the SWOM operation to each image path. Line 1 iterates through the list of raw image paths and reads each picture in as a grayscale image on line 2. The image is passed to the SWOM function on line 3 along with the **win_size**. The **win_size** argument determines the size of the window that SWOM uses. Since the design considerations establish that SWOM should use a window size of 32 or 64, this function requires the size to be passed in. Line 4 replaces **raw_pics** with **masked_pics** in the file path so that line 5 writes it to the appropriate video folder in the **masked_pics** folder. The next listing explains the **sliding_window_mask()** function that applies the SWOM operation.

```
1 def sliding_window_mask(img, win_size=64):
2     move_speed = win_size
3     x1 = 0
4     y1 = 0
5     x2 = win_size
6     y2 = win_size
7     vert_steps = math.ceil((len(img)-win_size)/move_speed)
8     hor_steps = math.ceil((len(img[0])-win_size)/move_speed)
```

Listing III.21: sliding_window_mask() function

The code in Listing III.21 implements a sliding window algorithm that calls the custom Otsu's Method function on each window. Note in line 2 that **move_speed** equals **win_size** so that no windows overlap. Lines 3-6 set up the first window with top left coordinate **(x1, y1) = (0,0)** and bottom right coordinate **(x2, y2)=(win_size, win_size)**. This first window covers the top left corner of the image. Line 7 calculates how many times the window must step across the image vertically, and line 8 calculates how many times the window must step across the image horizontally. Since the image is a square, these should be the same.

```
1     for y in range(vert_steps + 1):
2         for x in range(hor_steps + 1):
3             sub_img = img[y1:y2, x1:x2]
4             masked = mask_image(sub_img)
5             img[y1:y2, x1:x2] = masked
6             x1 = x1 + move_speed
7             x2 = x2 + move_speed
8         x1 = 0
9         x2 = win_size
10        y1 = y1 + move_speed
```

```
11          y2 = y2 + move_speed
12      return img
```

Listing III.22: sliding_window_mask() function

With the step numbers calculated, iterating across them with the nested for loop slides the window across the image as seen in Listing III.22. Lines 1 and 2 iterate across each horizontal and vertical step combination to cover the entire image. Line 3 extracts the window from the original image and passes it to the custom Otsu's Method **mask_image()** function on line 4. Line 5 stores the result of this in spot of the original window in the original image. Line 6 and 7 move the window to the right without changing the vertical coordinate which slides the window horizontally. When the window reaches the right edge of the image, lines 8 and 9 reset the window to the left edge of the image and lines 10 and 11 move the window down one window size. The process of horizontally sliding the window repeats until line 12 returns the result. This result replaces every window in the original image with its result from the **mask_image()** function.

```
1  def mask_image(img):
2      gray = img
3      if (len(gray.shape) == 3):
4          gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
5
6      kernel = np.array([[0, 0, 1, 0, 0],
7                         [0, 1, 1, 1, 0],
8                         [1, 1, 1, 1, 1],
9                         [0, 1, 1, 1, 0],
10                        [0, 0, 1, 0, 0]], dtype=np.uint8)
11     gray = cv2.GaussianBlur(gray, (5, 5), 0)
12     gray = cv2.morphologyEx(gray, cv2.MORPH_CLOSE, kernel)
```

```
13      gray = cv2.dilate(gray, kernel, iterations=1)
14      ret, mask = cv2.threshold(gray, 0, 255, cv2.THRESH_BINARY + cv2.
    THRESH_OTSU)
15      return mask
```

Listing III.23: mask_image() function

The **mask_image()** function listed in Listing III.23 augments OpenCV's implementation of Otsu's Method with the findings from Section 3.2.5. Lines 2-4 ensure the image is in grayscale format while lines 6-10 setup the circular kernel for the closing and dilating operations. This function performs the Gaussian blur, close, dilate, and Otsu's Method operations in lines 11, 12, 13, and 14 respectively then returns the final result back to the **sliding_window_mask()** function that calls it.

This point in the code is reached from the **convert_raw_pics_to_masked_pics()** function. This function passes the path of every saved raw picture to **sliding_window_mask()** which in turn calls **mask_image()** on each window of an image. The **mask_image()** function returns to **sliding_window_mask()** when it finishes processing a window. The **sliding_window_mask()** function returns to **convert_raw_pics_to_masked_pics()** when it finishes processing an image. The **convert_raw_pics_to_masked_pics()** function then returns to **create_test_dataset()** after it applies SWOM to all the raw images. Since the **convert_raw_pics_to_masked_pics()** function is the last function that **create_test_dataset()** calls, **create_test_dataset()** also returns back to the first function **create_test_and_train_data()** detailed in Listing III.8. This function then executes its second and last function call to **create_train_dataset()**.

The **create_train_dataset()** function transforms, rotates, and slices a single image of the iPhone LiDAR light pattern into many images to simulate what the pattern looks like projected at various angles in a real 3D environment. This saves the trouble

of having to manually record training data like in the case of the test data. It also provides a robust data generation framework for other direct LiDAR sensors besides the iPhone 12 Pro Max.

```
1 def create_train_dataset(path, num_transforms=9, num_rotations=3,
    win_size=64, img_size=512):
2     train_data_path = path + "\\train\\"
3     # Creates folders recursively if they do not exist:
4     # path\train\
5     # path\train\lidar\
6     # path\train\neg\
7     os.makedirs(train_data_path + 'lidar', exist_ok=True)
8     dst = path+'\\train\\neg'
```

Listing III.24: create_train_dataset() function

Similar to **create_test_dataset()**, Listing III.24 shows that **create_train_dataset()** accepts **win_size** and **img_size** arguments. However, it also accepts a **num_transforms** argument to control the number of affine transformations and a **num_rotations** argument to control how many times each affine transformation is rotated. Although not used directly, these arguments are passed forward to other functions until needed.

Line 2 in Listing III.24 adds 'train' to the path to specify the train folder since the path argument passed to this function just contains **path**. The **path\train** folder contains a simpler file structure than the test folder as depicted in the comments in lines 3-6. Therefore, line 7 immediately creates the **path\train\lidar** folder and line 8 defines the path to the **path\train\neg** folder.

```
1     try:
2         shutil.rmtree(dst)
3     except OSError as e:
```

```
4           pass
5       if 'light' in path:
6           shutil.copytree(path + '\\test\\masked_pics\\light\\neg', dst)
7       else:
8           shutil.copytree(path + '\\test\\masked_pics\\dark\\neg', dst)
```

Listing III.25: create_train_dataset() function

Lines 1 and 2 in Listing III.25 use the defined path to delete the **path\train\neg** folder if it exists. Lines 3 and 4 ignore any errors that arise if the folder does not exist. The train dataset uses the neg images from the test dataset as its own neg images. Therefore, lines 5-8 copy the neg images from the test dataset to the train dataset neg folder. If the path argument passed to this function contains the word 'light', that means this function was passed the path **path_light**. As declared earlier, this path name indicates that the videos were captured in a 35 Lux environment. This research labels these videos **light.avi** and **light_neg.avi** according to the defined video naming standard. Therefore, the **create_test_dataset()** function stores the SWOM result of the neg images in **path\test\raw_pics\light\neg**. For this reason, line 6 copies the test dataset neg folder from **path\test\raw_pics\light\neg** to the train dataset neg folder **path\train\neg**. Line 6 performs the same operation for the path **path** that contains videos **dark.avi** and **dark_neg.avi**.

```
1       pic = path + '\\light_pattern.png'
2       img = cv2.imread(pic, cv2.IMREAD_GRAYSCALE)
3       normalized = normalize_image(img)
4       normalized = cv2.resize(normalized, (img_size, img_size))
5       permute(normalized, train_data_path, num_transforms, num_rotations,
        win_size, img_size)
```

Listing III.26: create_train_dataset() function

Line 1 in Listing III.26 defines the path to the **path\light_pattern.png** image created by **create_test_dataset()**. Line 2 reads this LiDAR light pattern image as a grayscale image and passes it to the **normalize_image()** function on line 3. Once normalized, line 4 resizes the image to size **(img_size, img_size)** based on the received **img_size** argument. This function then passes the resized image along with the rest of the received arguments to the permute function on line 5. The following listings explain the **normalize_image()** function then move on to the **permute()** function.

```
1 def normalize_image(img):
2     gray = img
3     if (len(img.shape) == 3):
4         gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
5
6     kernel = np.array([[0, 0, 1, 0, 0],
7                        [0, 1, 1, 1, 0],
8                        [1, 1, 1, 1, 1],
9                        [0, 1, 1, 1, 0],
10                       [0, 0, 1, 0, 0]], dtype=np.uint8)
11    gray = cv2.GaussianBlur(gray, (5, 5), 0)
12    gray = cv2.morphologyEx(gray, cv2.MORPH_CLOSE, kernel)
13    ret, mask = cv2.threshold(gray, 0, 255, cv2.THRESH_BINARY + cv2.
    THRESH_OTSU)
14    true_coords = get_corners(mask)
```

Listing III.27: normalize_image() function

The **normalize_image()** function seen in Listing III.27 ensures that the LiDAR light point pattern image is square before it is used to generate training data. Lines 2-13 perform the same operation as the **mask_image()** function detailed in List-

ing III.23 minus the dilate operation. This creates a binary image mask that covers only the light points in the original image. Line 14 uses this mask to calculate the four corners of the minimum area quadrilateral that surrounds all the LiDAR light points in the original image. This finds the coordinates of each corner of the LiDAR light pattern, but the corners do not form a rectangle.

```
1     desired_coords = np.array([[680, 252], [1256, 252], [1256, 828],
      [680, 828]], dtype=np.float32)
2     M = cv2.getPerspectiveTransform(true_coords, desired_coords)
3     transformed_gray = cv2.warpPerspective(gray, M, (len(gray[0]), len(
      gray)))
4     transformed_mask = cv2.warpPerspective(mask, M, (len(gray[0]), len(
      gray)))
5     masked = cv2.bitwise_and(transformed_gray, transformed_mask)
6     return masked[252:828, 680:1256]
```

Listing III.28: normalize_image() function

With the corners of the LiDAR light pattern located, Listing III.28 affine transforms the pattern into a square. Line 1 defines the desired coordinates that form a (576, 576) square. This size comes from the frequency pattern classifier research explained in Section 2.5.1. However, the square gets resized to (512, 512), so any square larger than this is suitable since it does not lose quality when downsized. With both coordinate sets defined, line 2 calculates the matrix that affine transforms an image from the calculated **true_coords** of the LiDAR pattern to the desired coordinates of the square. Line 3 and 4 affine transform the original grayscale image and the binary mask respectively into squares. Line 5 applies **transformed_mask** from line 4 as a mask to **transformed_gray** from line 3 to isolate the LiDAR light points from any background noise. Line 6 then returns the square image containing only the LiDAR

light points from the original gray image.

This explains the transformation to a square image, but it does not explain how **get_corners()** finds the corners of the LiDAR light point pattern. The **get_corners()** function first calls **get_bbox()**, so the following listings first explain **get_bbox()** followed by **get_corners()**.

```
1 def get_bbox(img):
2     gray = img
3     if (len(img.shape) == 3):
4         gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
5     contours, _ = cv2.findContours(gray, cv2.RETR_EXTERNAL, cv2.
    CHAIN_APPROX_SIMPLE)
6     join_cnts = np.concatenate(contours)
7     return cv2.boundingRect(join_cnts)
```

Listing III.29: get_bbox() function

The **get_bbox()** function listed in Listing III.29 ensures the image is in grayscale in lines 2-4 then finds all the contours in the image on line 5. Line 6 concatenates the contours into a single object and finds the minimim bounding box of this object on line 7. A contour is the edge of an object. In the LiDAR light point image, a contour is a circle around one light point. A bounding box is the minimum area rectangle that encapsulates all the contours. Therefore, this function returns the smallest rectangle that encapsulates all the LiDAR light points in the image.

Figure 24 shows different bounding boxes for the square LiDAR light pattern image. Notice that the global bounding box in Figure 24(a) encapsulates all of the light points. However, it does not accurately reflect the corners of the LiDAR shape because the bottom left corner of the box must align with the top left corner to form a rectangle. Finding the local bounding boxes of each corner as depicted in Figure 24(b)

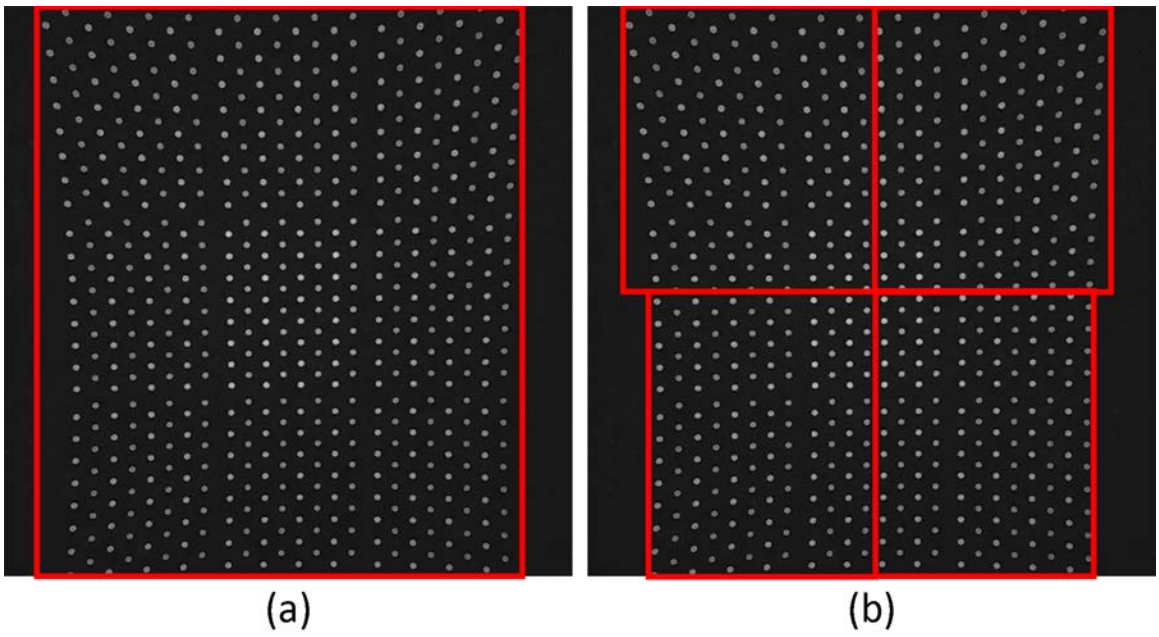

Figure 24: (a) Global vs. (b) local bounding boxes

solves this. The outside corner of each local bounding box represents the coordinates of the minimum area bounding quadrilateral.

```
1 def get_corners(img):
2     og_x, og_y, og_w, og_h = get_bbox(img)
3     cropped = img[og_y:(og_y + og_h), og_x:(og_x + og_w)]
```

Listing III.30: get_corners() function

The function **get_corners()** listed in Listing III.30 finds the coordinates of the minimum area bounding quadrilateral. Line 1 finds the global bounding box of the LiDAR light pattern image. Line 2 uses the global bounding box coordinates to extract the portion of the image that contains the LiDAR light points.

```
1     split_multiplier = 4
2     split_height = int(len(cropped) / split_multiplier)
3     split_width = int(len(cropped[0]) / split_multiplier)
```

```
4       bot_split = split_height * (split_multiplier - 1)
5       right_split = split_width * (split_multiplier - 1)
6
7       top_left = cropped[:split_height, :split_width]
8       top_right = cropped[:split_height, right_split:]
9       bot_left = cropped[bot_split:, :split_width]
10      bot_right = cropped[bot_split:, right_split:]
```

Listing III.31: get_corners() function

In Listing III.31, lines 1-5 calculate the values needed to split the extracted image in half vertically and horizontally. Lines 7-10 then divide the extracted image into four quadrants.

```
1       x, y, w, h = get_bbox(top_left)
2       tl = [x, y]
3       x, y, w, h = get_bbox(top_right)
4       tr = [x + w + right_split, y]
5       x, y, w, h = get_bbox(bot_left)
6       bl = [x, y + h + bot_split]
7       x, y, w, h = get_bbox(bot_right)
8       br = [x + w + right_split, y + h + bot_split]
9       cropped_coords = [tl, tr, br, bl]
10      true_coords = np.array([[i + og_x, j + og_y] for [i, j] in
    cropped_coords], dtype=np.float32)
11      return true_coords
```

Listing III.32: get_corners() function

In Listing III.32, lines 1-8 calculate the local bounding box for each quadrant. Line 9 aggregates the outside corner coordinates of each local bounding box to create the four coordinates of the minimum area bounding quadrilateral. These are the

coordinates in the cropped image, so line 10 adds offsets to find the true coordinates in the original image that it returns in line 11.

These coordinates are returned to **normalize_image()** which use them to affine transform the original image into a square as established in Listing III.27. The **normalize_image()** function returns the now square LiDAR light point image to the **create_train_dataset()** function which resizes it to the desired image size and passes it to the **permute()** function.

The **permute()** function immediately passes the image to **affine_transforms()**, **get_patches()**, and then **validate_patches()**. Therefore, the following listings describe these three functions and then uses them to describe permute(). To describe **affine_transforms()**, the following listing first describes its helper function **affine_transform()**.

The **affine_transform()** function ingests an image and three positional arguments (**i**, **j**, and **split**) to generate and return two affine transformations of the image. The **split** value represents the relative offset of an affine transformation that modifies the position coordinates of the affine transform provided in **i** and **j**.

Note that Figure 25(a) squeezes the bottom left corner of the original image towards the top left corner while Figure 25(b) squeezes the top right corner of the original image to the top left corner. Instead of affine transforming the image in all four directions, the image is affine transformed in two directions and then rotated to exploit the diagonal symmetry of the pattern.

The example in Figure 25 results from a **num_transforms** of 16. Therefore, Figure 25 represents the first of 16 affine transforms. All 16 positions are marked in Figure 26 with the first 7 points labelled. Note how points 1 and 2 fall on the squeeze points of the first affine transformation. Rather than squeezing to points 1 and 2, the second affine transformation squeezes to points 1 and 3, then 1 and 4, then and

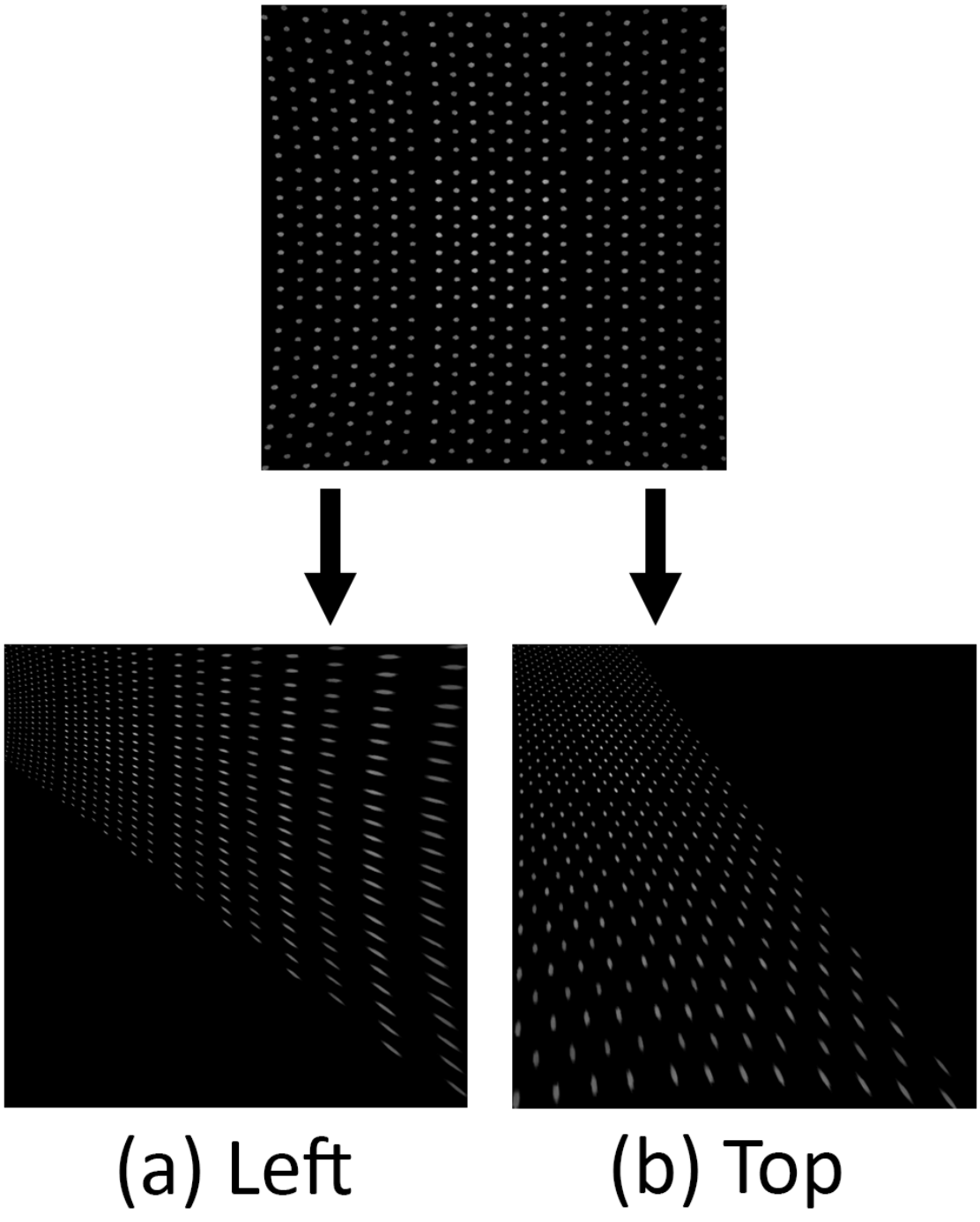

Figure 25: Affine_Transform() operation

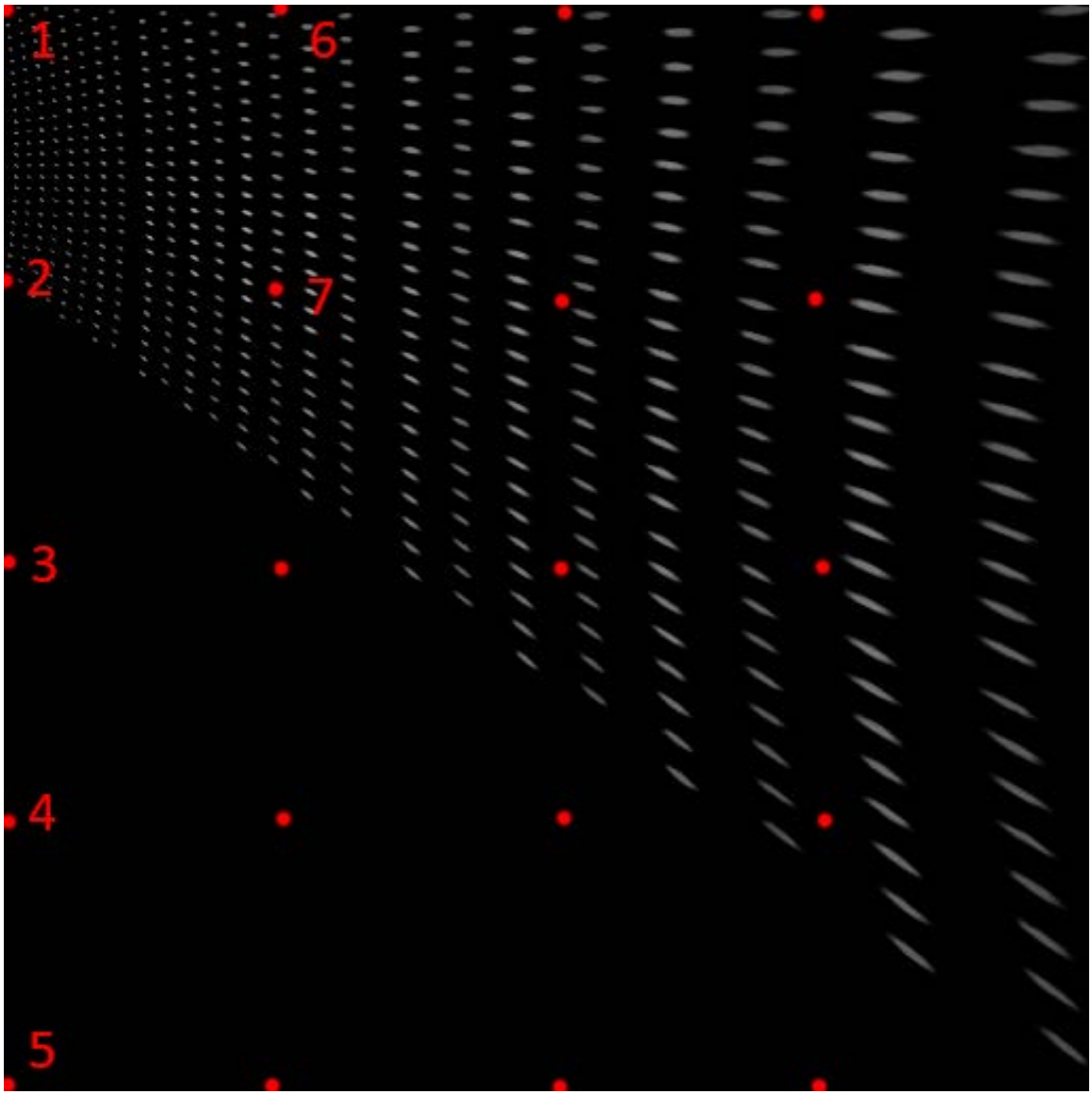

Figure 26: Affine transformation positions

5. Once the column is exhausted, the points slide over to the next column and start with an affine transformation to points 1 and 7. The **i** coordinate represents the top row point number while the **j** coordinate represents the column point number. The **split** is the distance between each vertical point. It multiplies **i** and **j** to find their actual positions in the image.

This is how the left transformation works. The top transformation works similarly, except one point always stays in the left most column while the second point iterates across the rows from the right. Again, the affine transformations are developed this way because exploiting the diagonal symmetry through rotation means that this covers just as many possibilities as affine transforming in all four directions. The following listing presents this concept in the **affine_transform()** function.

```
1 def affine_transform(img, i, j, split):
2     true_coords = np.array([[0, 0], [len(img[0]) - 1, 0], [len(img[0]) -
      1, len(img) - 1], [0, len(img) - 1]], dtype=np.float32)
3     tl = [i * split, j * split]
4     tr = true_coords[1]
5     br = true_coords[2]
6     bl = [i * split, (j + 1) * split]
7     desired_coords = np.array([tl, tr, br, bl], dtype=np.float32)
8    M = cv2.getPerspectiveTransform(true_coords, desired_coords)
9     left_dst = cv2.warpPerspective(img, M, (len(img[0]), len(img)))
```

Listing III.33: affine_transform() function

Listing III.33 shows the first half of the **affine_transform()** function. Line 2 saves the initial image corner coordinates for use in calculation of the affine transform matrix. Lines 3-9 perform the left affine transformation seen in Figure 25(a). Since this is the left transformation, the original top right and bottom right coordinates are saved in lines 4 and 5. However, lines 3 and 6 multiply the **i** and **j** coordinates by the

split to find their positions in the image. Line 7 aggregates these four coordinates into an array of the desired transformation coordinates. Line 8 calculates the required affine transformation matrix, and line 9 applies the matrix to the image to perform the left affine transformation.

```
1       tl = [j * split, i * split]
2       tr = [(j + 1) * split, i * split]
3       br = true_coords[2]
4       bl = true_coords[3]
5       desired_coords = np.array([tl, tr, br, bl], dtype=np.float32)
6      M = cv2.getPerspectiveTransform(true_coords, desired_coords)
7       top_dst = cv2.warpPerspective(img, M, (len(img[0]), len(img)))
8       return left_dst, top_dst
```

Listing III.34: affine_transform() function

Lines 1-9 in Listing III.34 perform the top affine transformation seen in Figure 25(b). This is the same operation as the left transformation, except the bottom right and bottom left coordinates do not change. The affine transformation instead shifts the top left and top right coordinates in line 7. Line 8 returns both the left and top transformation to the **affine_transforms()** function.

```
1 def affine_transforms(img, num_transforms):
2       transformed_imgs = []
3       rot_img = imutils.rotate(img, 90)
4       transformed_imgs.append(img)
5       transformed_imgs.append(rot_img)
```

Listing III.35: affine_transforms() function

Listing III.35 shows the beginning of the **affine_transforms()** function that performs all of the affine transformations. Before performing any transformations, lines

2-5 save the square LiDAR light pattern and its 90 degree rotation to the images list. The **affine_transforms()** functions returns this list, so appending the original image and its rotation ensures that the neural network trains on them in addition to the affine transformations.

```
1     if num_transforms > 1:
2         split = int(len(img) / np.sqrt(num_transforms))
3         for i in range(int(np.sqrt(num_transforms))):
4             for j in range(int(np.sqrt(num_transforms))):
5                 left_dst, top_dst = affine_transform(img, i, j, split)
6                 transformed_imgs.append(left_dst)
7                 transformed_imgs.append(top_dst)
8     return transformed_imgs
```

Listing III.36: affine_transforms() function

The rest of the **affine_transforms()** function is shown in Listing III.36. A **num_transforms** of value 1 returns the original image, so line 1 ensures that **num_transforms** is greater than 1. Line 2 calculates the proper **split** for the affine transform. Lines 3 and 4 iterate across the possible values for coordinates **i** and **j** derived from the provided **num_transforms**. Line 5 passes the LiDAR light pattern image, **i**, **j**, and **split** to the **affine_transform()** function, Lines 6 and 7 save the resulting left and top transformations to the return list. Line 7 returns the list of transforms to the **permute()** function which then passes them to the **get_patches()** function.

```
1 def get_patches(imgs, img_size):
2     imgs = np.expand_dims(imgs, -1)
3     patches4 = GetPatches(num_patches=4)(imgs)
4     patches4 = tf.reshape(patches4, [-1, patches4.shape[-3], patches4.
      shape[-2], 1])
```

```
5       patches16 = GetPatches(num_patches=4)(patches4)
6       patches16 = tf.reshape(patches16, [-1, patches16.shape[-3],
        patches16.shape[-2], 1])
7       imgs = np.squeeze(imgs, axis=-1)
8
9       patches = []
10      for patch in patches4.numpy():
11          res = cv2.resize(patch, (img_size, img_size))
12          patches.append(res)
13      for patch in patches16.numpy():
14          res = cv2.resize(patch, (img_size, img_size))
15          patches.append(res)
16      patches = np.array(patches)
17      patches = np.concatenate((imgs, patches), axis=0)
18      return patches
```

Listing III.37: get_patches()

The **get_patches()** function seen in Listing III.37 passes the images it receives to the **GetPatches** layer defined in Layers.py. Line 3 splits each image in quadrants. The quadrants are passed to the **GetPatches** layer again in line 5 which splits each quadrant into its own quadrants. Lines 2, 4, 6, and 7 consist of reshaping operations that properly format the data. The resulting patches are smaller than the original images since they are just a subsection, so lines 9-17 resize them to match the size of the images they were cut from and concatenates them all together. Each image input to this function returns 21 images (1 original + 4 patches + 16 patches) of the original size. These images are returned to the **permute()** function which then passes them to the **validate_patches()** function.

```
1 def validate_patches(patches):
2     valid_patches = []
3     for patch in patches:
4         ret, mask = cv2.threshold(patch, 0, 255, cv2.THRESH_BINARY + cv2
      .THRESH_OTSU)
5         contours, _ = cv2.findContours(mask, cv2.RETR_EXTERNAL, cv2.
      CHAIN_APPROX_SIMPLE)
6         num_dots = len(contours)
7         if num_dots > 16:
8             join_cnts = np.concatenate(contours)
9             x, y, w, h = cv2.boundingRect(join_cnts)
10            bbox_area = w * h
11            total_area = len(patch) * len(patch[0])
12            coverage = bbox_area / total_area
13            if coverage > 0.50:
14                valid_patches.append(patch)
15
16    return np.array(valid_patches)
```

Listing III.38: validate_patches()

Listing III.38 shows the **validate_patches()** function. This function checks all the images created by **get_patches()** to ensure the neural network only trains on images that contain a significant amount of LiDAR light. Lines 2-6 count the contours in an image to identify how many LiDAR light points exist in the image. If the image contains more than 16 points at line 7, it then calculates the cumulative area of the light points in lines 8-12. If line 13 indicates that the light points account for more than 50 percent of the image, it adds the image to the valid patch list at line 14. This valid list is returned to the **permute()** function.

```
1 def permute(img, path, num_transforms, num_rotations,win_size,img_size):
2     transformed_imgs = affine_transforms(img, num_transforms)
3     patches = get_patches(transformed_imgs, img_size)
4     valid_patches = validate_patches(patches)
```

Listing III.39: permute()

The first half of the **permute()** function seen in Listing III.39 receives the normalized image of the LiDAR light points in the **img** argument. It also receives the path containing the **path\train** folder. The function affine transforms the normalized image in line 2 based on the **num_transforms** argument, creates patches in line 3, and validates the patches in line 4.

```
1     write_count = 0
2     for patch in valid_patches:
3         rotations = []
4         for r in range(num_rotations):
5             rotated = imutils.rotate(patch, random.randint(0, 360))
6             rotations.append(rotated)
7         for pic in rotations:
8             noisy = random_noise(pic, mode='gaussian', var=0.05 ** 2)
9             noisy = (255 * noisy).astype(np.uint8)
10            mask = sliding_window_mask(noisy, win_size)
11            cv2.imwrite(path + 'lidar\\' + str(write_count) + '.png',
    mask)
12            write_count += 1
```

Listing III.40: permute()

The rest of the **permute()** function is shown in Listing III.40. Lines 2-6 randomly rotate each patch the number of times directed by **num_rotations**. The **normalize()** function removes all the background noise when it isolates the LiDAR light points,

so lines 8 and 9 add back a small, random amount of Gaussian noise to emulate real world conditions. Line 10 then applies SWOM to each rotation and saves the result to the **path\train\lidar** folder. Each image is saved as **<number>.png** with <number> counting up from 0.

After **permute()** saves all the images, it returns to **create_train_dataset()** which returns to **create_test_and_train_data()**. This concludes the explanation of how Preprocess_Footage.py generates a training dataset from a single image and extracts a test dataset from the videos recorded by Capture_Footage.py. The complete Preprocess_Footage.py script is listed in Appendix B for reference.

3.4.5 Neural Network Layers Python Script

The neural network Python script Train_Evaluate_NN.py builds a neural network with layers from Layers.py. Therefore, this section first explains the contents of Layers.py then describes how Train_Evaluate_NN.py uses the layers to build, train, and test a neural network.

Layers.py contains three TensorFlow neural network layer classes: GetPatches, GetBatchCosts, and ApplyWeights. The following listings describe each layer.

```
1 class GetPatches( tf.keras.layers.Layer ):
2     def __init__(self, num_patches):
3         super(GetPatches, self).__init__()
4         self.num_patches = tf.constant(num_patches, dtype=tf.float32)
5     def build(self, input_shape):
6         self.batch_size = input_shape[0]
7         self.patch_size = tf.cast(tf.math.divide(input_shape[1], tf.math
    .sqrt(self.num_patches)), dtype=tf.int32)
```

Listing III.41: GetPatches

Although **GetPatches** is used in Listing III.37 to split training data images, the class primarily serves as a processing layer in the neural network. Line 1 in Listing III.41 indicates that the **GetPatches** class extends the **tf.keras.layers.Layer** class which allows it to easily integrate into the neural network. When an instance of this class is constructed with a **num_patches** argument, the class constructor shown in lines 2-4 stores the **num_patches** argument to **self.num_patches**. The **build()** function on lines 5-7 uses **self.num_patches** to calculate and store the patch size in **self.patch_size**. The **build()** function also stores **self.batch_size** based on the input shape since the size of the input's first axis is the batch size. Most classes have constructors and callable functions, but the build() function exists in between the two. A TensorFlow neural network model calls the **build()** function after it constructs an instance of the class but before it calls the **call()** function. As the name implies, this typically occurs when building the neural network which allows the layer to store metadata such as desired patch size for later use.

```
1     def call(self, inputs):
2         sizes = [1, self.patch_size, self.patch_size, 1]
3         patches = tf.image.extract_patches(images=inputs,
4                                            sizes=sizes,
5                                            strides=sizes,
6                                            rates=[1, 1, 1, 1],
7                                            padding='VALID')
8         return tf.reshape(patches, [self.batch_size, tf.cast(self.
    num_patches, dtype=tf.int32), self.patch_size, self.patch_size, 1])
```

Listing III.42: GetPatches

Listing III.42 shows the last function in the **GetPatches** layer class. The **call()** function uses both **self.num_patches** and **self.patch_size** from the **init()** and **build()**

functions respectively. The **call()** function uses the patch size on line 2 to calculate the **sizes** array it passes to **tf.image.extract_patches()** on lines 3-7. It also uses the number of patches and patch size to properly shape the output of the patches returned to the caller on line 8. If the images are evenly divisible by the patch size, then the function **tf.image.extract_patches()** evenly divides a collection of images into patches of that size. The next layer, **GetBatchCosts**, is a tad more complicated, so it is described in multiple sections.

```
1 class GetBatchCosts(tf.keras.layers.Layer):
2     def __init__(self, ring_size=1):
3         super(GetBatchCosts, self).__init__()
4         self.ring_size = tf.constant(ring_size, dtype=tf.float64)
5
6     def call(self, inputs):
7         inputs_shape = tf.shape(inputs)
8         inputs = tf.reshape(inputs, [inputs_shape[0], inputs_shape[1],
    inputs_shape[2]])
9         inputs_shape = tf.shape(inputs)
```

Listing III.43: GetBatchCosts

Before diving into the heavy math portion, this paragraph first describes the simple parts of the **GetBatchCosts** class as seen in Listing III.43. Similar to the **GetPatches** class, this class also starts with an **init()** function. Instead of a mandatory patch number argument, it takes an optional **ring_size** argument which it stores in **self.ring_size**. By default, it sets the width of the frequency pooling rings to 1 pixel. This class also does not have a build() layer. It gets all the information it needs at construction and run time. At run time, the **call()** function reshapes the inputs and stores the new input shape on lines 7-9. The input shape is a 4-Dimensional (4D) tensor in the shape [**batch_size, image_width, image_height, pixel_intensity**].

Each pixel intensity only contains one integer, so line 8 erases the unnecessary pixel intensity axis and instead stores their values in the [**batch_size, image_width, image_height**] tensor.

```
1         tf_fft = tf.abs(tf.signal.fftshift(tf.signal.fft2d(tf.cast(
    inputs, dtype=tf.complex64)))) + tf.constant(0.001)
2         tf_fft_shape = tf.shape(tf_fft)
3         ef = tf.reshape(tf_fft, [tf_fft_shape[0], tf_fft_shape[1],
    tf_fft_shape[2]])
4         st = tf.sparse.from_dense(ef)
```

Listing III.44: GetBatchCosts

Listing III.44 displays the next portion of the **call()** function in the **GetBatchCosts** layer which performs two important function. First, it calculates the magnitude of the 2D DFT of the images on line 1. This line also adds a small constant value to each element of the resulting 2D DFT tensor because TensorFlow sparse tensors do not keep track of elements that contain the value 0. Rather than maintaining the data of every single value in a tensor, the sparse tensor maintains a list of indices along with a matching list of the non-zero values at those indices. This function needs to track every element of the tensor, so it adds a small constant to ensure no index contains 0. Lines 2 and 3 once again erase the unnecessary pixel value axis and convert the tensor to a sparse tensor in line 4.

```
1         st_indices = tf.cast(tf.identity(st.indices), dtype=tf.float32)
2         batch_nums, ef_idxs = tf.split(st_indices, [1, 2], axis=1)
3         batched_ef_idxs = tf.reshape(ef_idxs, [inputs_shape[0], -1, 2])
```

Listing III.45: GetBatchCosts

Since the sparse tensor tracks the values of a 3D tensor, line 1 in Listing III.45 returns a list of 3D indices of the format [**image identification number, 2**] where the second dimension contains the horizontal and vertical location of each pixel. Since this stores all pixels for an image on one axis, the image identification number indicates which image it belongs to. However, a more convenient format is a list where each element in the first dimension contains every pixel for a single image in the format [**number of images, number of pixels, 2**]. In this format, the first axis has one independent element for each image, the second axis stores every pixel for a single image, and the third axis now contains the horizontal and vertical location coordinates. To accomplish this, line 2 strips off the image identification number axis and line 3 reshapes the leftover horizontal and vertical location values.

```
1         ef_shape = tf.shape(ef)
2         mid_val = tf.divide(tf.subtract(ef_shape[-2], 1), 2)
3         mid = tf.cast(tf.repeat(mid_val, repeats=[2]), dtype=tf.float32)
```

Listing III.46: GetBatchCosts

The **GetBatchCosts** layer **call()** function continues in Listing III.46. With the sparse tensor now properly formatted, this part of the function calculates the Chebyshev distance of each pixel. Section 2.5.1 defines the iterative approach to this, but this process can also be implemented with linear algebra. Lines 1-3 calculate the middle pixel value **mid_val** such that the middle pixel can be located by the index **(mid_val, mid_val)**. It creates the **mid** tensor in the shape [**1, 2**] where both elements contain the value **mid_val**. Only the 2D location of the middle pixel is required because this function calculates the pixel distances for only one image. All of the images are the same size, so it reuses the calculated distances for all the images.

```
1         first_batched_ef_idxs = tf.expand_dims(batched_ef_idxs[0], 0)
2         sub = tf.subtract(first_batched_ef_idxs, mid)
3         abs_t = tf.abs(sub)
4         maxxed = tf.reduce_max(abs_t, axis=2)
5         floored = tf.floor(maxxed)
6         floored = tf.cast(floored, dtype=tf.float64)
7         div = tf.divide(floored, self.ring_size)
8         dist = tf.cast(div, dtype=tf.int32)
```

Listing III.47: GetBatchCosts

Recall that the previous steps in Listing III.45 reshape the sparse tensor indices to the format [**number of images, number of pixels, 2**]. Line 1 in Listing III.47 selects a single image from the sparse tensor and adds an axis to the beginning which results in a single image with the shape [**1, number of pixels, 2**]. This results in an ordered list of pixel indices for a single image that can be used to calculate the distance to the middle pixel. Line 2 subtracts **mid** from this list of pixel indices. Since the last axes of the pixel indices and **mid** are both of shape 2, it broadcasts the **mid** tensor and subtracts the **mid_val** from each pixel index. Lines 3-7 then perform linear algebra on the subtraction result to find the tensor **dist**. This tensor has the shape [**1, number of pixels**] and contains an ordered list of integer distances that correspond to specific pixels. These distances are used to sum the pixel values of each ring.

```
1         num_rings = tf.cast(tf.divide(tf.math.ceil(tf.divide(ef_shape
    [-2], 2)), self.ring_size), dtype=tf.int32)
2         res_inputs = tf.transpose(tf.reshape(ef, [ef_shape[0], -1]))
3         summed_rings = tf.transpose(tf.math.unsorted_segment_sum(
    res_inputs, dist[0], num_rings))
4         return summed_rings
```

Listing III.48: GetBatchCosts

Listing III.48 presents the final part of the **GetBatchCosts** layer **call()** function. Line 1 calculates the total number of rings based on the given ring size and shape of the images. Line 2 reshapes the output from the 2D DFT to the format [**number of images, number of pixels per image**]. The first axis is a list of images and each image is a list of pixel intensity values. These lists of pixel intensity values are in the exact same order as the **dist** tensor. Therefore, each list of pixel intensity values can be compared to the **dist** tensor to categorize the pixel values based on their corresponding value in **dist**. The value in **dist** is each pixel values' ring number. For example, every pixel intensity value at a location that corresponds to the distance 2 in the **dist** tensor all belong to ring 2. Line 3 uses the **tf.math.unsorted_segment_sum()** function to sum the pixel intensities based on their corresponding ring number in **dist**. This function takes **res_inputs** of pixel intensity values and sums them together based on their corresponding distance value in **dist**[**0**]. Recall that **dist** is shaped [**1, number of pixels, 2**], so **dist**[**0**] returns the tensor of shape [**number of pixels, 2**]. The function also accepts **num_rings** to ensure the number of rings does not exceed the number of possible distances. Line 3 then transposes the final result to return a tensor of shape [**1, number of rings**] that contains a single value for each ring which consists of the sum of every pixel intensity value in that ring. Line 4 returns these sums to the caller.

This concludes the explanation of the **GetBatchCosts** layer implementation. The linear algebra implementation of the frequency pooling layer has two major benefits. First, linear algebra can process batches of images much faster than iterating across each image individually. Second, implementing this layer exclusively in TensorFlow linear algebra operations allows TensorFlow to automatically calculate the gradient and update the trainable weights via SGD in the following ApplyWeights layer.

```
1  class ApplyWeights(tf.keras.layers.Layer):
2      def __init__(self):
3          super(ApplyWeights, self).__init__()
4
5      def build(self, input_shape):
6          self.w = self.add_weight(
7              shape=(input_shape[1], ),
8              initializer=tf.keras.initializers.RandomUniform(minval=0,
       maxval=0.3),
9              trainable=True,
10             constraint=tf.keras.constraints.NonNeg(),
11             name='ring_weights'
12         )
13
14     def call(self, inputs):
15         return tf.multiply(inputs, self.w)
```

Listing III.49: ApplyWeights

Similarly to the **GetPatches** layer, Listing III.49 shows that the **ApplyWeights** layer is fairly simple. This class exclusively performs the job of the frequency filter described in Section 2.5.1. The **GetBatchCosts** layer does the heavy lifting of pooling the pixel intensity values into a tensor with shape [**1, number of rings**], so all this layer has to do is maintain an equivalent number of weights and multiply them together. Lines 6-11 initialize a list of weights with the shape of the inputs' second axis to ensure it contains one weight value for each ring sum. Line 8 initializes each weight with a small random value between 0.0 and 0.3 and marks them as trainable in line 9 so SGD iteratively updates them to decrease loss and improve classification accuracy. Line 10 prevents the possibility of negative weights. All of the details in the **build()** function build this layer in accordance with the initial research that this is

derived from [11]. This concludes the explanation of Layers.py. The Layers.py script is included in Appendix C for reference. With the different layer classes defined, the following section explains how Train_Evaluate_NN.py uses them.

3.4.6 Train and Evaluate Neural Network Python Script

The final Python script that supports this research, Train_Evaluate_NN.py, builds the model with layers from Layers.py then trains and tests it on the datasets produced by Preprocess_Footage.py. The following sections explain the different parts of the Train_Evaluate_NN.py Python script.

```
1  def get_model(img_size, batch_size, ring_size):
2      inputs = tf.keras.Input(shape=(img_size, img_size, 1), batch_size=
       batch_size)
3      patches_4 = GetPatches(num_patches=4)(inputs)
4      patches_16 = GetPatches(num_patches=16)(inputs)
5      costs1 = GetBatchCosts(ring_size=ring_size)(inputs)
6      costs4 = GetBatchCosts(ring_size=ring_size)(patches_4)
7      costs16 = GetBatchCosts(ring_size=ring_size)(patches_16)
8      costs = tf.keras.layers.concatenate([costs1, costs4, costs16], axis
       =1)
9      weighted_costs = ApplyWeights()(costs)
10     dense1 = tf.keras.layers.Dense(512, activation='relu', trainable=
       True)(weighted_costs)
11     sigmoid = tf.keras.layers.Dense(1, activation='sigmoid')(dense1)
12     model = tf.keras.Model(inputs=inputs, outputs=sigmoid)
13     return model
```

Listing III.50: get_model()

Listing III.50 shows the most important function in this script. The **get_model()** function builds the neural network model according to the specifications in Section 3.3.3. This model consists of three main parts: slicing the images into 4 patches

which are sliced into 16 patches, pooling the frequency values with ring width of 1 pixel, and training 512 ReLU neurons. The input layer on line 2 passes the images in parallel to a **GetPatches** layer on line 3 to split them into 4 patches as well as a **GetPatches** layer on line 4 to split them into 16 patches. The initial inputs and the resulting patches are all concurrently passed to separate **GetBatchCosts** layers on lines 5-7. All three **GetBatchCosts** layers pass their summed ring values to a concatenate layer on line 8 which combines them all into a single tensor. The concatenate layer passes this tensor to the **ApplyWeights** layer on line 9 which multiplies all the summed ring values by the layers' trainable weights. It passes this product to the Dense layer on line 10 consisting of 512 ReLU neurons which apply their trainable weights and biases to the tensor. This layer then passes its results to the final Dense layer of a single sigmoid neuron that classifies whether or not the image contains LiDAR light points. The layers described in this paragraph are also presented graphically in Figure 27.

```
1 def get_dataset(pics_path, img_size, batch_size):
2     data_dir = pathlib.Path(pics_path)
```

Listing III.51: get_dataset()

With the neural network created, datasets must next be generated to train and test it. The **get_dataset()** function displayed in Listing III.51 accepts a file path, image size, and batch size as arguments. Based on the neural network design, image size remains at 512 and batch size remains at 1. Therefore, this function just needs the path to a folder that contains LiDAR and neg images. This function uses the **tf.keras.utils.image_dataset_from_directory()** utility function to create datasets. This utility function looks for folders containing images related to the name of the folder. This is why the Preprocessing_Footage.py script stores all training data in

separate **lidar** and **neg** folders nested within the train folder. When the dataset utility function looks through the train folder, it classifies all the images in the **lidar** folder as LiDAR images and all the images in the **neg** folder as neg images. Line 2 in this function formats the train image directory to pass to the utility function.

```
1    train_ds = tf.keras.utils.image_dataset_from_directory(data_dir,
2                          validation_split=0.2,
3                          shuffle=True,
4                          subset='training',
5                          color_mode='grayscale',
6                          seed=random.randint(0, 10000),
7                          image_size=(img_size, img_size),
8                          label_mode='binary')
```

Listing III.52: get_dataset()

The dataset utility function splits the training set into a final training dataset **train_ds** and a validation dataset **val_ds**. In Listing III.52, **data_dir** contains the path **path\train**. Therefore, the utility function creates **train_ds** from images in **path\train\lidar** and **path\train\neg**. Line 2 indicates that this function saves 20% of the images for the validation set, so **train_ds** consists of 80% of the images. Line 3 randomly shuffles the 80% of images. Line 4 labels this dataset as the training dataset. Line 5 tells the function to read the images in grayscale. Line 6 initializes the seed for randomly shuffling the images. Line 7 sets the size that the images are resized to if they are not already that size. Finally, line 8 indicates this is a binary dataset with only two classes.

```
1    val_ds = tf.keras.utils.image_dataset_from_directory(data_dir,
2                        validation_split=0.2,
3                        subset='validation',
```

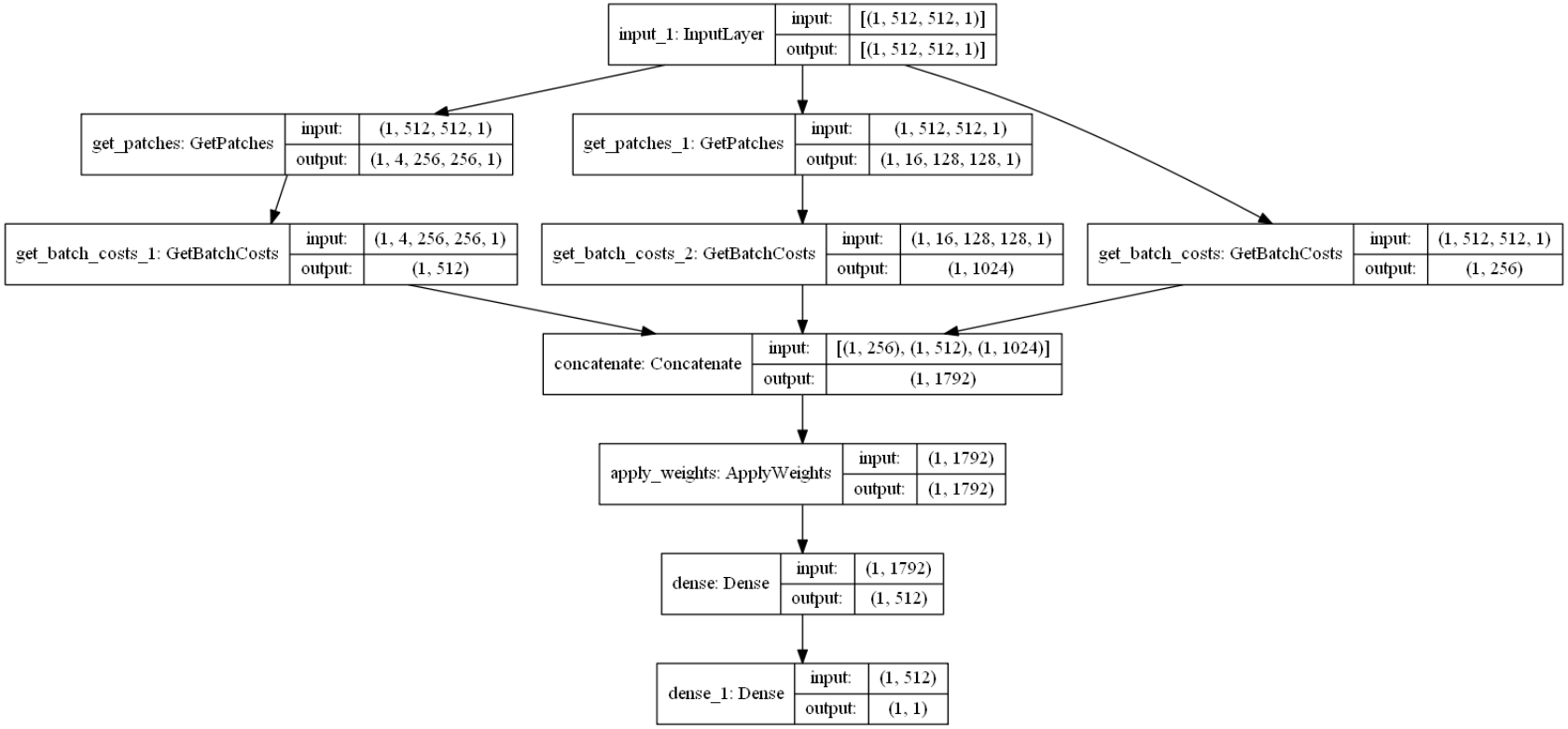

Figure 27: Neural network graph

```
4                       shuffle=True,
5                       color_mode='grayscale',
6                       seed=random.randint(0, 10000),
7                       image_size=(img_size, img_size),
8                       label_mode='binary')
9     return train_ds, val_ds
```

Listing III.53: get_dataset()

The second execution of this utility function in Listing III.53 creates the validation dataset **val_ds**. It creates the dataset exactly the same as the **train_ds** dataset, except this dataset only gets 20% of the images. Line 9 returns both the final training dataset and the validation dataset to the caller function.

```
1 def train_and_save_model(path, img_size=512):
2     batch_size = 1
3     ring_size = 1
4     model = get_model(img_size, batch_size, ring_size)
5     train_ds, val_ds = get_dataset(path + '\\train', img_size,
      batch_size)
6     model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate
      =0.00001),
7                   loss=tf.keras.losses.BinaryCrossentropy(),
8                   metrics=[tf.keras.metrics.BinaryAccuracy()])
9     model.fit(train_ds, validation_data=val_ds, batch_size=batch_size,
      epochs=5)
10    model.save(path + '\\model')
```

Listing III.54: train_and_save_model()

As the name implies, the **train_and_save_model()** function shown in Listing III.54 trains the neural network and saves the trained variables to the hard drive for later use. The default settings used for all function calls are **batch_size=1** and **ring_size=1**

based on the neural network design. The **num_classes** is always set to 2 since the only two classes are **lidar** and **neg**. Lines 1-3 instantiate these default settings. Line 4 calls **get_model()** to build the model. Assuming Preprocess_Footage.py already generated the training dataset images, line 5 calls **get_dataset()** to split this dataset into the final training and validation datasets. Line 6-8 compile the model into a complete neural network object that can be trained and evaluated. It compiles the model with the SGD Adam algorithm that uses a learning rate of 0.00001. There are a lot of strong opinions on learning rates, but this research finds that this value works well across the datasets. It also compiles the model with the BinaryCrossentropy() function that measures the loss and uses Adam to update the trainable variables. Finally, it compiles the model with the BinaryAccuracy() metric to report how accurately the model classifies whether or not LiDAR light points exist in an image. Executing model.fit on line 9 uses the training dataset to train the neural network for 5 epochs. At the end of each epoch, the classification accuracy of the model is evaluated on the validation dataset. The small amount of epochs achieves above 90% training accuracy nearly every time, so this proves to be a good balance between accuracy and training speed. It trains the neural network on **batch_size** images at a time which is left at the value 1. Finally, line 10 saves the entire trained model which includes the final values of the trainable variables in the **ApplyWeights** and Dense ReLU layers. It also includes the layers themselves so it can be loaded on another computer with no knowledge of the source code. After this function trains and saves the model, the final step is to load and evaluate its performance on real images of LiDAR light.

```
1 def load_and_evaluate_model(path, img_size=512):
2     batch_size = 1
3     tests = path + '\\test\\masked_pics'
```

```
4       test_paths = []
5       test_names = []
6       for test in os.listdir(tests):
7           test_paths.append(os.path.join(tests, test))
8           test_names.append(test)
```

Listing III.55: load_and_evaluate_model()

As shown in Listing III.55, the **load_and_evaluate_model()** loads a saved model and evaluates the accuracy of classification on real LiDAR images. Similar to the training process, the model is evaluated on **batch_size** images at a time. Line 2 sets the **batch_size** to 1. Recall that Capture_Footage.py records videos of LiDAR light points which Preprocess_Footage.py processes and saves in **path\test\masked_pics**. This is the path defined on line 3. Lines 4-8 iterate through this folder to find all the existing folders. Every folder in the **masked_pics** folder represents one dataset of real world LiDAR images. These test datasets evaluate the accuracy of the neural network trained on simulated LiDAR images.

```
1       model = tf.keras.models.load_model(path + '\\model')
2       results = []
3       for test_path, test_name in zip(test_paths, test_names):
4           dataset = tf.keras.utils.image_dataset_from_directory(test_path,
5                                       color_mode='grayscale',
6                                       shuffle=True,
7                                       image_size=(img_size, img_size),
8                                       batch_size=batch_size,
9                                       label_mode='binary')
10
11
12          accuracy = model.evaluate(dataset, verbose=0)
13          results.append(test_name + ' - Accuracy: ' + str(accuracy[1]))
```

```
14
15      return results
```

Listing III.56: load_and_evaluate_model()

Line 1 in Listing III.56 loads the trained neural network from the model folder and evaluates it on every folder found in **masked_pics**. Line 4 uses the dataset utility function to create a dataset from a given folder in **masked_pics**. The utility function shuffles the images in the dataset, reads them as grayscale images, and resizes them to the desired image size. The binary label mode makes sure the dataset class labels do not clash with the binary class labels of the neural network. Line 12 evaluates the loaded model for the given **masked_pics** folder and saves the results to a list on line 13. After the for loop evaluates the model on every available test dataset, line 15 returns the results of how accurately the model classifies the images in each data set.

This concludes the explanation of the Train_Evaluate_NN.py Python script. The entire script is included in Appendix D for further reference.

3.5 Tools

3.5.1 Python Development Environment

The Python development environment consists of many tools that support this research. Table 3 lists the versions of the Python interpreter, PyCharm IDE, Conda and pip package managers, and main packages used in this research. The versions of the other minor packages mentioned in Section 2.6.1 are subject to the Python version.

3.5.2 Apple Development Environment

The Apple development environment primarily relies on Xcode Version 12.5 build 12E262. This IDE builds an iPhone app from source code and deploys it to an iPhone.

Table 3: Python environment

Name	Version
Python	3.9.7
PyCharm	PC-212.4746.96
Conda	4.10.3
Pip	21.2.4
OpenCV	4.5.2
TensorFlow	2.6.0
NumPy	1.21.2
Scikit-Imag	0.18.3
Imutils	0.5.4

Xcode runs on a MacBook Pro to deploy the 3D scanner app to the iPhone when connected via lightning cable. When Xcode is running with the 3D scanner app project (pointCloudSample) loaded, a project tab appears at the top of the Xcode window as seen in Figure 28. If the iPhone is connected to the MacBook, the "No devices connected to 'My Mac'" text shown in the red box instead shows the iPhone when the project tab is clicked. Select the iPhone and then click the arrow just to the left of the project tab to deploy the 3D scanner app to the iPhone. Once deployed, the iPhone retains the app for later use even when unplugged from the MacBook.

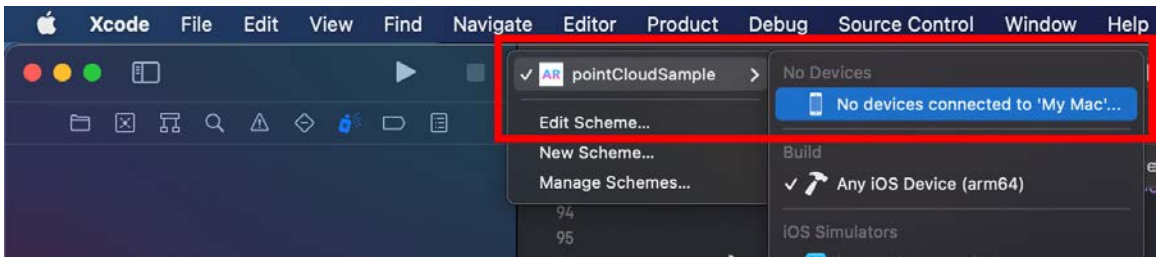

Figure 28: Xcode deploying 3D scanning app

3.6 Devices

The following sections detail the devices used in this research. They describe the devices that make up the camera infrastructure, the cellphones and their respective

applications, and the computers that process the video footage and deploy the iPhone app.

3.6.1 Camera Infrastructure

Table 4 lists the model of the security camera, switches, and web plug in. The security camera connects to the PoE switch, the PoE switch connects to the switch, and the switch connects to the desktop workstation. This allows the desktop workstation to access the camera with the NACL Web Plug-in software to edit the camera settings.

Table 4: Camera infrastructure devices

Device Type	**Name**	**Version**
Security Camera	Lorex LNB8921B [23]	N/A
PoE Switch	POE-SW502 [94]	N/A
Switch	TE100-S5/A v3.2R [93]	N/A
Software	NACL Web Plug-in [49]	1.0.0.1

3.6.2 Cellphones

This research uses the three cellphones listed in Table 5. The design considerations described in Section 3.2.1 identify that both the Galaxy S20 Ultra 5G and the Huawei P40 Pro contain indirect ToF LiDAR sensors, so only the direct ToF rear LiDAR sensor on the iPhone 12 Pro Max is used in the majority of this research. Note that the iPhone's "Kernel Version" in Table 5 refers to its build number instead.

Table 5: Cellphone specifications

Name	**Model**	**OS Version**	**Kernel Version**
iPhone 12 Pro Max	MGCJ3LL/A	iOS 14.4	18D52 (Build)
Galaxy S20 Ultra 5G	SM-G988UZAAXAA	Android 10.0	4.19.81-18024814
Huawei P40 Pro	ELS-NX9	Android 10.0	4.14.116)

3.6.3 Cellphone Applications

This research uses 3D scanning applications to project LiDAR light from all three phones as well as a light meter application on the iPhone to measure the light level in the environment. The Photone light meter app (version 2.7.1) simply requires a user to place a small piece of standard A4 paper over the front facing camera to diffuse incoming light and measure the light level [62]. Xcode is used to load a 3D scanning application posted on Apple's developer website directly to the iPhone [98]. The Galaxy S20 Ultra 5g and Huawei P40 Pro download the 3D Live Scanner app from their respective app stores [6][60]. The iPhone app produces the LiDAR pattern seen in Figure 3 while the 3D Live Scanner apps on both phones produces the LiDAR pattern seen in the Figure 4.

3.6.4 Computers

The Python scripts run on the desktop workstation that connects to the security camera. The desktop workstation has the specifications seen in Table 6.

Table 6: Desktop Specifications

Hardware	**Specification**
OS	Windows 10 Home version 10.0.19043 Build 19043
Processor	Intel Core i7 8700 3.2GHz
GPU	NVIDIA GeForce GTX 1070 4095MB
RAM	16GB 1069MHz
Motherboard	ASUS Prime Z370-A
Storage	250GB Samsung SSD 850 EVO

The 2020 MacBook Pro 13-inch (Model A2251) deploys the 3D scanning app to the iPhone. It has the specifications listed in Table 7.

Table 7: Laptop Specifications

Hardware	**Specification**
OS	Big Sur version 11.5
Processor	2GHz Quad-Core Intel Core i5
GPU	Intel Iris Plus Graphics 1536MB
RAM	32GB 3733MHz LPDDR4X
Motherboard	ASUS Prime Z370-A
Storage	1 TB SSD

3.7 Design Summary

This chapter explores the design considerations that influence choices made in the system design. The resulting system design is both summarized and explained is detail. Following the detailed explanation, the tools and devices that support this research are presented. The next chapter presents specific research methodology used to properly evaluate this system.

IV. Methodology

4.1 Problem/Objective

This research develops a system to detect and identify cellphone LiDAR light in security camera footage. This system answer the following questions:

- Can a security camera reliably capture cellphone LiDAR light?
- Is there a reliable method to identify captured cellphone LiDAR light in a video?
- What factors most heavily influence the identification of cellphone LiDAR light in a video?

To identify how well the system answers these questions, it is used to test the following hypotheses:

1. In an environment with controlled textures and light levels, a security camera configuration exists that captures cellphone LiDAR light projected into the environment the majority of the time.
2. If video footage of cellphone LiDAR light is captured in an environment with controlled textures and light levels, each frame of the video can be processed to identify whether or not it contains cellphone LiDAR light with an accuracy above 50%.

The first hypothesis proves whether or not the security camera reliably captures LiDAR light to ensure that it can gather suitable data to evaluate the accuracy of the neural network.

The System Under Test is defined in Section 4.2. This is followed by descriptions of the experiment metrics in Section 4.3, experiment factors in Section 4.4, uncontrolled variables in Section 4.5, and constant parameters in Section 4.6. Section 4.7

then describes the experiments followed by a description of how they are statistically analyzed in Section 4.8.

4.2 System Under Test

As described in Chapter III, the System Under Test (SUT) generates a training dataset from a single LiDAR light pattern image and trains a neural network to detect similar patterns in real LiDAR light images captured by a security camera. The trained neural network is evaluated on its classification accuracy and the amount of time it takes to train and test it. The SUT diagram in Figure 29 illustrates the design as well as how the metrics, factors, variables, and parameters detailed in following sections relate to the design.

4.2.1 Assumptions

Since the security camera must enable night mode to capture LiDAR light, this research assumes that it records footage in a low light environment. A low light environment is simulated with a 1000 Lumen LED light bulb that illuminates the test surface to 35 Lux.

This research also assumes that the environment consists exclusively of Lambertian surfaces that can diffuse LiDAR light for the security camera to capture. This assumption focuses on how the LiDAR light pattern appears in simple environments without the need to consider a wide range of potential distortion caused by Spectral surfaces.

This research assumes that experiments 1, 2, and 3 are performed in order as well as one directly after the other. Experiments 2 and 3 therefore exclude the setup steps listed in experiment 1.

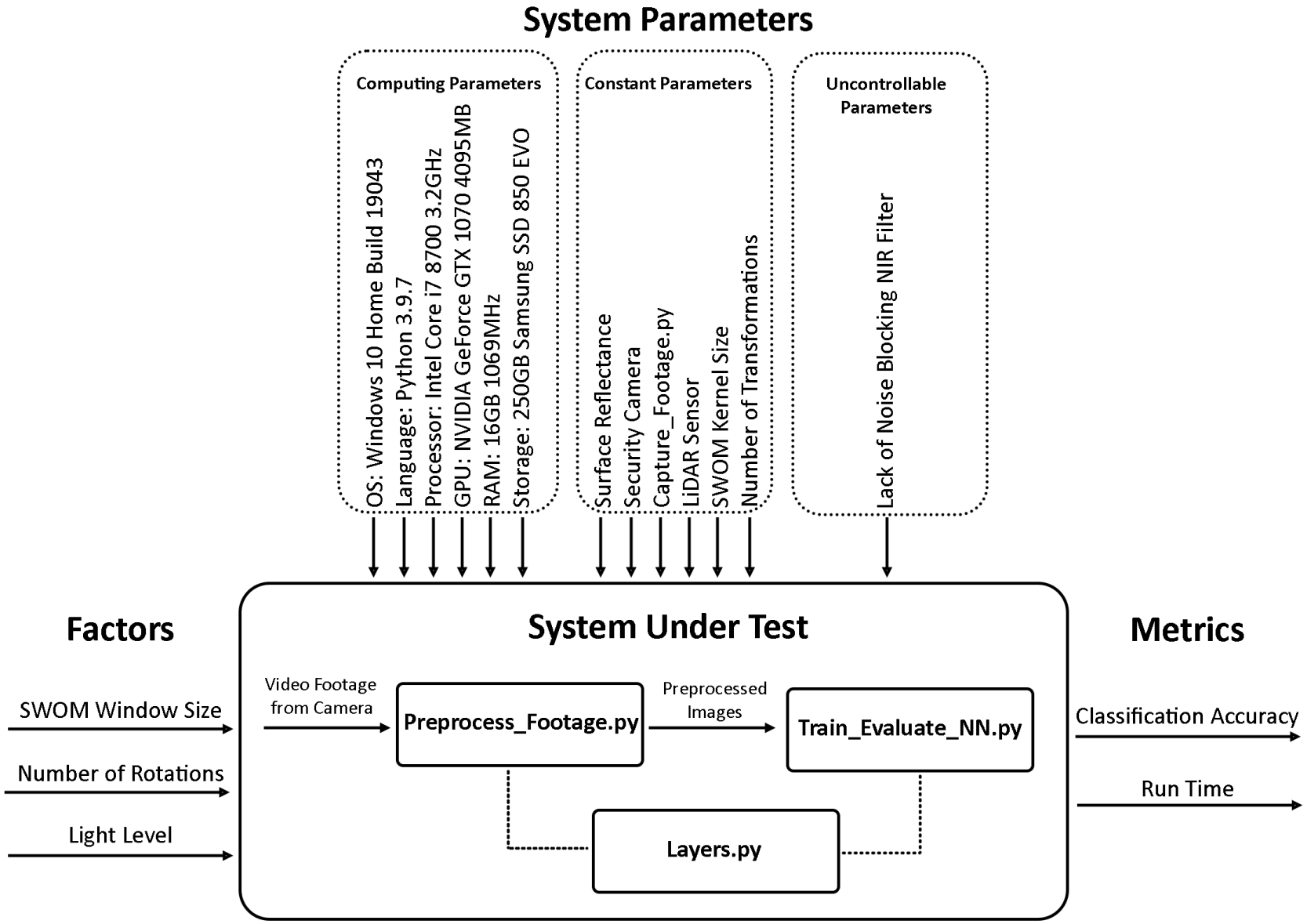

Figure 29: System Under Test Diagram

4.3 Metrics

The neural network is evaluated on its classification accuracy and run time. The value ranges for both of these metrics are listed in Table 8.

- Classification Accuracy: This metric evaluates the neural network based on its number of correct classifications and incorrect classifications. This is defined formally as

$$Acc = \frac{TP + TN}{TP + FP + TN + FN} \tag{31}$$

  A True Positive (TP) represents the correct classification of a LiDAR image. A True Negative (TN) represents the correct classification of a negative image. A False Positive (FP) represents the incorrect classification of a negative image as a LiDAR image. A False Negative (FN) represents the incorrect classification of a LiDAR image as a negative image. The accuracy value is a ratio, so it does not have any units. Higher accuracy is better.

- Run Time: The time it takes to generate a test and training dataset, train the neural network with the training dataset, and evaluate the neural network on the test dataset for a given experiment. The Python scripts use the Python 'time' library to measure the execution time in seconds with nanosecond accuracy [48]. Lower run time is better.

Table 8: Experiment metrics

Metric	**Unit**	**Expected Range**
Classification Accuracy	N/A	$0.5 < Acc \leq 1.0$
Execution Time	seconds	0 - 300 s

4.4 Factors

Experiment 3 analyzes how two different levels of the three factors listed in Table 9 affect the neural network classification accuracy and run time.

- SWOM Window Size: This measures the pixel width of the square window used to perform the Sliding Window Otsu's Method (SWOM) operation in the Preprocessing_Footage.py Python script. SWOM is used to process both the training and test data, so this factor heavily affects the classification accuracy and run time of the neural network. In experiments where this is not the target factor, it is set to 64 pixels.

- Number of Rotations: When generating the LiDAR images for the training dataset, the Preprocessing_Footage.py Python script randomly rotates each transformation this number of times. Therefore, this factor directly influences both the quality and volume of the training data which in turn affects the classification accuracy and run time of the neural network. In experiments where this is not the target factor, it is set to 3 rotations.

- Light Level: The assumptions limit the light level in the environment to a maximum of 35 Lux, but there is a still a big visual difference between 0 Lux and 35 Lux. The light level affects how the SWOM isolates the LiDAR light point pattern in test video footage. Although this does not impact the quality of the training dataset, it heavily affects the quality of the test dataset. In experiments where this is not the target factor, it is set to 0 Lux.

Table 9: Experiment factors

Factor	Levels	Description
Window Size	32 pixels 64 pixels	Width of square window used to perform SWOM on images
Number of Rotations	3 rotations 9 rotations	Number of random rotations generated for image transformation
Light Level	0 Lux 35 Lux	Illumination level of environment

4.5 Uncontrolled Variables

The most prevalent uncontrollable variable takes the shape of noise captured in the camera footage. NIR filters block NIR light to decrease noise in an image. Disabling this filter increases noise in the video footage which is unavoidable in the pursuit of capturing LiDAR light.

4.6 Experiment Parameters

Multiple different parameters affect the accuracy and run time of the experiments, so they are held constant across all experiments to prevent them from influencing the results. Table 10 lists the parameters and their constant values.

- Surface Reflectance: The reflectance of a surface influences the brightness of reflected LiDAR light which influences the intensity of LiDAR light recorded in video footage. Therefore, all video footage captures LiDAR light projected onto the same off-white painted wall in every experiment.

- Security Camera: The security camera's rolling shutter can effect the captured LiDAR light pattern. The camera capture settings listed in Table 2 influence the recorded video footage. The camera video settings, most notably the 15 FPS and 1080p resolution, also affect the captured footage and subsequently

the neural network performance. Overall, the security camera largely influences the classification accuracy and run time of the neural network. However, this research primarily focuses on how processing the video footage affects the neural network accuracy, so all security camera characteristics, as well the model of the camera itself, are kept constant.

- Capture_Footage.py: Similarly to the security camera, this Python script directly influences the recorded video footage. Regardless of the camera settings, this script controls the FPS and video codec of the video stored on disk. This script is a tool to record video footage for this research rather than an experiment factor itself, so the script settings are held constant to avoid influencing the performance of the neural network.

- LiDAR Sensor: This research explores how to detect and identify the pattern produced by a specific LiDAR sensor. This may be expandable to other 3D LiDAR sensors, but this research focuses a single pattern to evaluate classification accuracy. Therefore, only the iPhone 12 Pro Max LiDAR sensor is used for all experiments.

- SWOM Kernel Size - The SWOM kernel size is the size of the two kernels seen in the create_mask() function in Preprocess_Footage.py. This controls the circular kernel used in the morphological close and dilate operations as well as the square Gaussian blur kernel. This kernel size undoubtedly affects the classification accuracy of the neural network, but its affects relate to the size of the LiDAR light points which is a function of distance. This research does not explore the affect of distance on the LiDAR light, so the SWOM kernel stays at a constant size of 5 pixels.

- Number of Transformations: The number of transformations controls how many

affine transforms the Preprocessing_Footage.py Python script performs on the LiDAR light pattern image to simulate real world LiDAR projections. Although the various values of this parameter affect the metrics, this research focuses on how the number of rotations affects the metrics due the diagonal symmetry of the LiDAR light pattern. Therefore, this parameter is kept constant at a value of 16 transformations.

Table 10: Experiment parameters

Parameter	**Value**
Surface Reflectance	Off-White Wall
Capture_Footage.py	15 FPS, MJPG codec
Security Camera	Lorex LNB8921B
LiDAR Sensor	iPhone 12 Pro Max rear LiDAR sensor
SWOM Kernel Size	5 pixels
Number of Transformations	16

4.7 Experimental Design

Hypothesis 1 is tested by manually reviewing captured video footage to identify if the majority of frames contain LiDAR light as described in Section 4.7.1. If proven, hypothesis 2 is then tested as described in Section 4.7.2. If also proven, then the third experiment described in Section 4.7.3 analyzes the affects of the factors listed in Table 9.

4.7.1 Experiment 1

The first experiment tests hypothesis 1. If LiDAR light is captured in more than 50% of the video frames, the camera is considered capable of reliably capturing LiDAR light. This is tested with the following steps.

1. Connect the camera to the desktop workstation as instructed in Section 3.3.1. Configure the camera in accordance with Section 3.4.1.

2. Open the 3D scanner app on the iPhone to emit LiDAR light.

3. Position both the security camera and the iPhone 1 meter above the ground pointed perpendicularly to the wall. The security camera sensor should lie directly above the iPhone LiDAR sensor highlighted in Figure 30(a).

4. Turn off the light to ensure an illumination of 0 Lux.

5. On the desktop workstation, run the Capture_Footage.py Python script. When the live stream opens, press 'v' to capture 30 seconds of video footage.

6. In the same folder as the Preprocess_Footage.py script, create the folder structure path\test\videos\. Place the recorded video in the videos folder.

7. Run the Preprocess_Footage.py script. This saves each frame of the video to path\raw_pics\<video_name>\.

8. Manually inspect each image in the raw_pics folder to identify the percentage of frames that contain LiDAR light.

4.7.2 Experiment 2

The second experiment tests hypothesis 2. The accuracy of the trained neural network is evaluated with the following steps.

1. Capture 30 seconds of video with no LiDAR light and name the file 'dark_neg.avi'.

2. Capture 30 seconds of video footage containing LiDAR light according to the process described in Section 3.4.3 and name the file 'dark.avi'.

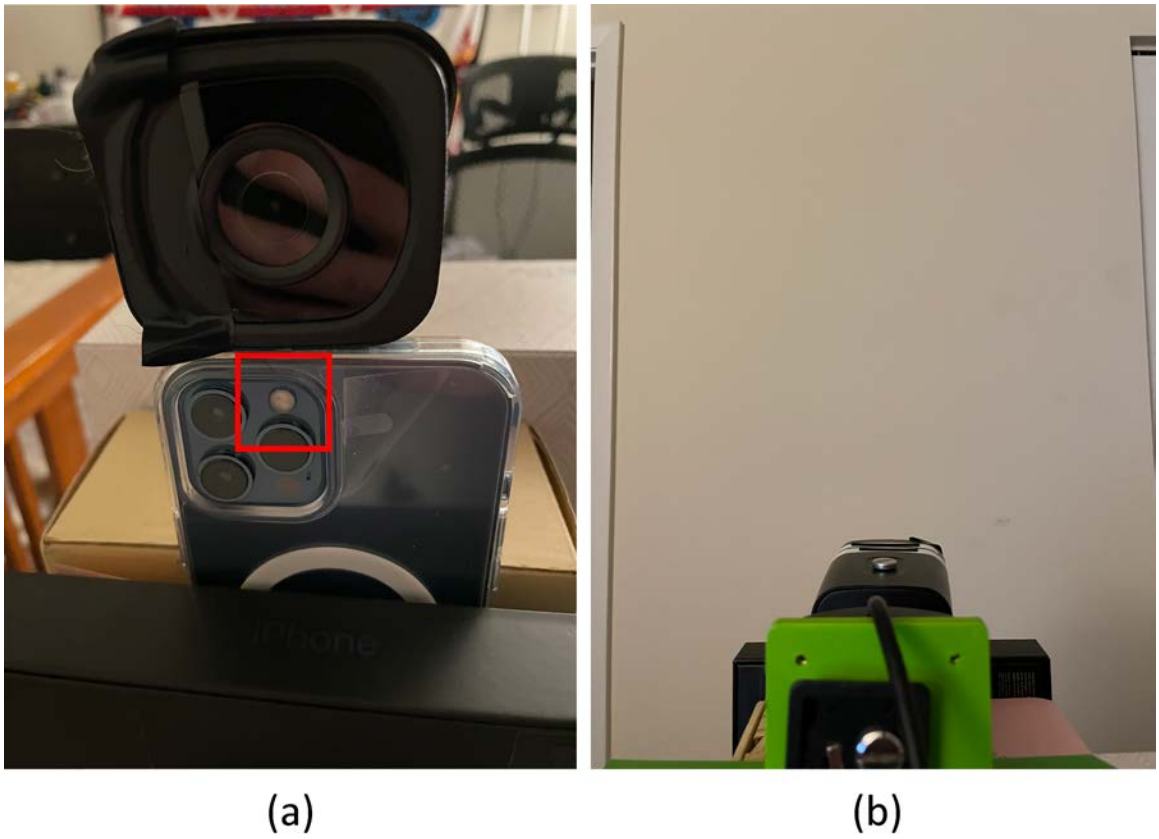

Figure 30: Security camera positioned above iPhone LiDAR sensor

3. In the same folder as the Preprocess_Footage.py and Train_Evaluate_NN.py scripts, create the folder structure path\test\videos\. Place 'dark.avi' and 'dark_neg.avi' in the videos folder.

4. Run the Preprocess_Footage.py script on path\.

5. Run the Train_Evaluate_NN.py Python script on path\.

6. Record the classification accuracy results from step 5 and the total run time of steps 4 and 5.

4.7.3 Experiment 3

The third experiment finds the levels of the factors defined in Table 9 that improve classification accuracy and run time. This experiment tests each level of each factor 10 times to ensure the statistical analysis detailed in Section 4.8 produces significant results. The factor level is chosen randomly at the start of each run. To test the SWOM window size and number of rotations, steps 3-5 of experiment 2 are repeated with only one of the factors modified at a time. To test the light level, the following steps are performed.

1. Turn off the light to ensure an environment of 0 Lux.
2. Capture 30 seconds of video with no LiDAR light and name the file 'dark_neg.avi'.
3. Capture 30 seconds of video footage containing LiDAR light according to the process described in Section 3.4.3 and name the file 'dark.avi'.
4. Create the folder structure path\test\videos\. Place 'dark.avi' and 'dark_neg.avi' in the videos folder.
5. Turn on the light to ensure an environment of 35 Lux.
6. Capture 30 seconds of video with no LiDAR light and name the file 'light_neg.avi'.
7. Capture 30 seconds of video footage containing LiDAR light according to the process described in Section 3.4.3 and name the file 'light.avi'.
8. Create the folder structure path_light\test\videos\. Place 'light.avi' and 'light_neg.avi' in the videos folder.
9. Run the Preprocess_Footage.py script on either path\ or light_path\.
10. Run the Train_Evaluate_NN.py Python script on same path as step 9.

11. Record the classification accuracy results from step 10 and the total run time of steps 9 and 10.

12. Repeat steps 9-11 ten times for path\ and ten times for light_path\. Choose path randomly, but do each exactly 10 times.

The Experiment.py Python script in Appendix E performs the trials for experiment 3. This script imports the functions that Preprocess_Footage.py and Train_Evaluate_NN.py call in their main() functions to emulate the steps described in experiment 3. This script saves text files with the classification accuracy and run time results for each level of each factor. The results from these files are manually copied into the Statistic_Analysis.py Python script provided in Appendix F to perform the statistical analysis.

4.8 Statistical Analysis

The results of experiments 1 and 2 are analyzed in a basic manner to prove or disprove the proposed hypotheses. However, if both are proven, then experiment 3 analyzes how different levels of different factors affect the classification accuracy and run time. The Wilcoxon signed-rank test is used in Statistic_Analysis.py to compare the two levels of each factor in experiment 3 since they cannot be assumed to be normally distributed. The Wilcoxon signed-rank test tests the null hypothesis that both samples come from the same distribution. The difference in mean and standard deviation between two samples indicates the likelihood that they come from the same distribution. If this likelihood is significantly small (< 0.05), the null hypothesis is rejected since they likely come from different distributions. Only a sample size of 6 is required to reach this this level of significance, but this research uses a sample size of 10 to increase the power of the test [99]. This test is the non-parametric version of the Student's T-test that compares normal distributions [100]. These comparisons

identify if the different factor levels significantly affect the neural network's classification accuracy and run time. If they do, Chapter V interprets the results to identify which factor levels produce the best overall metrics and why.

As explained in Section 2.6.1, the means and standard deviations are only as precise as the inputs. Therefore, they are precise up to 9 decimals for the run time and 16 decimals for the classification accuracy. However, the precision of the inputs does not affect the precision of the Wilcoxon signed-rank test since it is a rank test [100]. The Wilcoxon signed-rank test outputs results with 9 decimals of precision. The results of this research can be adequately analyzed in text with only 3 decimals of precision, so values are rounded to 3 decimals for use in text. However, the Python script in Appendix F contains the full precision accuracy and run time values. This script prints out the full precision mean, standard deviation, and Wilcoxon signed-rank test values with 9, 16, and 16 decimals of precision respectively.

4.9 Methodology Summary

This chapter presents the research goals and corresponding hypotheses followed by assumptions. It also presents a SUT diagram to illustrate the implementation of the system design and how it is used it to test the hypotheses. The SUT includes the metrics, factors, variables, and parameters that the sections following the SUT elaborate on. Three experiments are detailed to test the hypotheses and evaluate how the factor levels influence the metrics. Finally, a method for statistically analyzing the experiment results is presented.

V. Results and Analysis

5.1 Overview

This chapter presents the results of the first two experiments in Section 5.2. Section 5.3 presents the results from experiment 3 and statistically analyzes how the factor levels influence the classification accuracy and run time. Section 5.4 provides a conceptual analysis of the results the three experiments.

5.2 Hypotheses

The first experiment is performed according to Section 4.7.1. To identify a reliable camera configuration, the camera must record a video with LiDAR light in the majority of its frames. Configuring the camera according to Section 3.4.1 resulted in a video with LiDAR light in every frame. All 450 frames of the 30 second long video recorded at 15 FPS with 1/15 s shutter speed contain LiDAR light. This confirms that the camera can reliably capture LiDAR light which proves hypothesis 1.

The second experiment is performed according to Section 4.7.2. This experiment results in a classification accuracy of 0.916. Although this does not answer how different factors influence the classification accuracy, it does indicate that it is possible to develop an effective binary classifier to detect LiDAR light in images captured by security cameras. With both of the hypotheses proven, the next section analyzes the different factor level results from experiment 3.

5.3 Experiment 3

This experiment evaluates the effect of different factor levels on the classification accuracy of the neural network as well as the time it takes to generate the training and test datasets, train the neural network, and test the neural network as detailed in

Section 4.7.3. The Experiments.py Python script in Appendix E runs the experiments for each factor level and saves the results to a file which are then copy and pasted to the Statistical_Analysis.py Python script listed in Appendix F. This statistical analysis script calculates means and standard errors as well as the Wilcoxon signed-rank test results with floating point precision, but the results are truncated to 3 decimal places as explained in Section 4.8.

5.3.1 SWOM Window Size

20 trials of experiment 3 randomly choose a SWOM window size of either 32 or 64 pixels for a total of 10 trials for each window size. These trials keep the number of rotations constant at 3 rotations and the light level constant at 0 Lux. In the following analysis, recall that higher classification accuracies and lower run times are better.

Table 11 shows that the 64 pixel window size classifies images approximately 9% more accurately than the 32 pixel window size on average. Furthermore, the standard deviation of the 64 pixel window size classification accuracy is an order of magnitude smaller than the standard deviation of the 32 pixel window size. To compare these results, the null hypothesis H_0 states that both of these window sizes result in classification accuracies from the same distribution with a significance level $\alpha = 0.05$. The Wilcoxon signed-rank test reports a p-value of $p = 0.001$ which indicates that there is only a 0.1% chance that these classification accuracies come from the same distribution. Since $p < \alpha$, the null hypothesis is rejected which indicates that these SWOM window sizes result in significantly different classification accuracies. Since the 64 pixel window results in a higher mean classification accuracy than the 32 pixel window, the 64 pixel window significantly improves the classification accuracy over the 32 pixel window.

Table 11: window size classification accuracy

SWOM Window Size	Mean	Standard Deviation
32 pixels	0.905	0.03
64 pixels	0.997	0.004

Table 12 shows that the 64 pixel window size runs approximately 19 seconds faster than the 32 pixel window size on average. Both window sizes result in run times with fairly similar standard deviations. To compare these results, the null hypothesis H_0 states that both of these window sizes result in run times from the same distribution with a significance level $\alpha = 0.05$. The Wilcoxon signed-rank test reports a p-value of $p = 0.001$ which indicates that there is only a 0.1% chance that these run times come from the same distribution. Since $p < \alpha$, the null hypothesis is rejected which means that these SWOM window sizes result in significantly different run times. Since the 64 pixel window results in a lower mean run time than the 32 pixel window, the 64 pixel window significantly decreases the run time over the 32 pixel window.

Table 12: SWOM window size run time

SWOM Window Size	Mean	Standard Deviation
32 pixels	110.835 s	1.399 s
64 pixels	91.978 s	1.615 s

These results show that a 64 pixel SWOM window significantly increases the classification accuracy and decreases the run time as compared to the 32 pixel window. Therefore, the following experiments use a 64 pixel window to evaluate the effects that the number of rotations and light level have on the classification accuracy and run time.

5.3.2 Number of Rotations

20 trials of experiment 3 randomly choose 3 or 9 rotations for a total of 10 trials for each number of rotations. These trials keep the SWOM window size constant at 64 pixels and the light level constant at 0 Lux. In the following analysis, recall that higher classification accuracies and lower run times are better.

Table 13 shows that 3 rotations and 9 rotations result in very similar classification accuracy means and standard deviations. To compare these results, the null hypothesis H_0 states that both of these numbers of rotations result in classification accuracies from the same distribution with a significance level $\alpha = 0.05$. The Wilcoxon signed-rank test reports a p-value of $p = 0.556$ which indicates that there is a 55.6% chance that these classification accuracies come from the same distribution. Since $p > \alpha$, the null hypothesis is rejected which means that these numbers of rotations do not result in significantly different classification accuracies.

Table 13: Number of rotations classification accuracy

Number of Rotations	**Mean**	**Standard Deviation**
3 rotations	0.992	0.006
9 rotations	0.993	0.009

Table 14 shows that 3 rotations result in a run time that is approximately 50% shorter than the length of the run time produced by 9 rotations. However, both numbers of rotations result in run times with fairly similar standard deviations. To compare these results, the null hypothesis states that both of these numbers of rotations result in run times from the same distribution with a significance level $\alpha = 0.05$. The Wilcoxon signed-rank test reports a p-value of $p = 0.001$ which indicates that there is only a 0.1% chance that these run times come from the same distribution. Since $p < \alpha$, the null hypothesis is rejected which means that these numbers of rotations result in significantly different run times. Since 3 rotations result in a lower

mean run time than 9 rotations, 3 rotations significantly decrease the run time over 9 rotations.

Table 14: Number of rotations run time

Number of Rotations	Mean	Standard Deviation
3 rotations	89.672 s	1.144 s
9 rotations	187.577 s	1.276 s

These results show that the number of rotations does not significantly affect the classification accuracy, but it does significantly affect the run time. 3 rotations significantly decrease the run time, so the following experiment uses 3 rotations along with the 64 pixel SWOM window size.

5.3.3 Light Level

20 trials of experiment 3 randomly choose a 0 Lux or 35 Lux light level for a total of 10 trials for each light level. These trials keep the SWOM window size constant at 64 pixels and the number of rotations constant at 3 rotations. In the following analysis, recall that higher classification accuracies and lower run times are better.

Table 15 shows that the 0 Lux light level classifies images approximately 33% more accurately than the 35 Lux light level on average. Furthermore, the standard deviation of the 0 Lux light level classification accuracy is two orders of magnitude smaller than the standard deviation of the 35 Lux light level. To compare these results, the null hypothesis states that both of these light levels result in classification accuracies from the same distribution with a significance level $\alpha = 0.05$. The Wilcoxon signed-rank test reports a p-value of $p = 0.001$ which indicates that there is only a 0.1% chance that these classification accuracies come from the same distribution. Since $p < \alpha$, the null hypothesis is rejected which means that these light levels result in significantly different classification accuracies. Since the 0 Lux light level results in a

higher mean classification accuracy than the 35 Lux light level, the 0 Lux light level significantly improves the classification accuracy over the 35 Lux light level.

Table 15: Light level classification accuracy

Light Level	**Mean**	**Standard Deviation**
0 Lux	0.993	0.007
35 Lux	0.659	0.101

Table 16 shows that the 35 Lux light level runs approximately 30 seconds faster than the 0 Lux light level on average. Furthermore, the standard deviation of the 35 Lux light level run time is approximately 75% of the 0 Lux light level run time. To compare these results, the null hypothesis states that both of these light levels result in run times from the same distribution with a significance level $\alpha = 0.05$. The Wilcoxon signed-rank test reports a p-value of $p = 0.001$ which indicates that there is only a 0.1% chance that these run times come from the same distribution. Since $p < \alpha$, the null hypothesis is rejected which means that these light levels result in significantly different run times. Since the 35 Lux light level results in a lower mean run time than the 0 Lux light level, the 35 Lux light level significantly decreases the run time over the 0 Lux light level.

Table 16: 0 Lux vs. 35 Lux run time

Light Level	**Mean**	**Standard Deviation**
0 Lux	92.232 s	1.033 s
35 Lux	62.68 s	0.787 s

These results show that a 0 Lux light level significantly increases the classification accuracy, but a 35 Lux light level significantly decreases the run time. This research values classification accuracy much more than run time, so 0 Lux light performs the best. Although this 0 Lux light results in a run time that is 150% that of the 35 Lux light level run time, this only indicates a 30 second increase over the 35 Lux light run

time. Although it is relatively a large difference, a 1.5 minute run time in comparison to a 1 minute run time is well worth a classification accuracy increase of 30%.

The light results also show that classification capabilities are severely degraded in environments with more light. The 0 Lux light level is essentially pitch black except for the LiDAR light, so it is easier to isolate and identify the LiDAR points. Even at a modest light level of 35 Lux, the classification accuracy drops all the way down to 0.659. An untrained binary classifier can randomly guess to achieve a classification accuracy around 0.5, so an accuracy of 0.659 is not good at all.

5.3.4 Best Factor Levels

Considering only the two values for each of the three factors compared in this section, the best factor levels are a 64 pixel SWOM window, 3 rotations, and 0 Lux light level. These factor values result in an average classification accuracy of 0.997 and an average run time of 91.978 seconds.

5.4 Conceptual Analysis

The first two experiments prove the ability to reliably capture LiDAR light with a security camera and identify it with an accuracy above 50%. These proofs of the two hypotheses are necessary to indicate that the goal of cellphone LiDAR detection and identification is possible. This prevents any time being wasted analyzing factors in experiment 3. If LiDAR light cannot be captured and identified at a rudimentary level in the first two experiments, the effects of the different factor levels on the classification accuracy and run time cannot be adequately analyzed in experiment 3.

Experiment 3 identifies that a 64 pixel SWOM window, 3 rotations, and 0 Lux light level maximizes the classification accuracy and minimizes the run time without sacrificing the classification accuracy. Classification accuracy is not sacrificed for run

time because the goal of this research is to develop a framework that accurately detects and identifies LiDAR in video footage, not a framework that quickly detects and identifies LiDAR. The run times are useful as a secondary metric for choosing factor levels that do not result in significantly different accuracies, but the run times never take precedent over accuracy. Even in the consideration of run time as a primary metric, it can be shown that the combined accuracy and run time metric fails to properly measure the system performance.

Comparing the accuracies and run times of the two levels of each factor produces a relative measurement of either high, low, or same. For example, the 32 pixel SWOM window results in a relatively low accuracy and relatively high run time when compared to the 64 pixel SWOM window. This results in the accuracy-run time pairs of Low-High for the 32 pixel window and High-Low for the 64 pixel window. A high accuracy and low run time is definitely better than a low accuracy and high run time, so Low-High is labeled as Bad and High-Low is labeled as Good in Table 17. The number of rotations does not affect the classification accuracy, but it does affect the run time, so this definitively labels the low run time as Good and the high run time as Bad. The light level is what causes some issues, because a high accuracy is only definitively better than a low accuracy if it pairs with a low run time like in the window size factor. Therefore, Accuracy-to-Run Time ratio of each level must instead be compared to decide which performs better.

Using the mean accuracy and run time values from Table 15 and Table 16, the 0 Lux light level has an Accuracy-to-Run Time ratio of $0.993/92.232 = 0.011$ whereas the 35 Lux light level has an Accuracy-to-Run Time ratio of $0.659/62.68 = 0.011$. These results are the same since the accuracy and run times in the tables are rounded to 3 decimals, but the ratios are slightly different if the values are rounded to 4 decimals. Using accuracies and run times rounded to 4 decimals, the 0 Lux light level

has an Accuracy-to-Run Time ratio of 0.9931/92.2323 = 0.0107 whereas the 35 Lux light level has an Accuracy-to-Run Time ratio of 0.659/62.6798 = 0.0105. The ratio of the 0 Lux light level is approximately 0.0002 more than that of the 35 Lux light level, so the High-High light level pair performs better than the Low-Low pair in this case.

Table 17: Relative comparisons of factor level pairs

Factor	**Value**	**Accuracy Level**	**Run Time Level**	**Result**
SWOM Window Size	32 pixels	Low	High	Bad
	64 pixels	High	Low	Good
Number of Rotations	3	Same	Low	Good
	9	Same	High	Bad
Light Level	0 Lux	High	High	?
	35 Lux	Low	Low	?

This method of relative comparison defines a way to choose the better of two factor levels as long as they have different Accuracy-to-Run Time ratios, but this is not necessarily an effective method for this research. Given an accuracy of 0.99 with a 0.99 second run time and an accuracy of 0.01 with a 0.001 second run time, the Accuracy-to-Run Time ratio comparison reports that 0.99/0.99 = 1 is a worse performance ratio than 0.01/0.001 = 10. Although this is true based on the ratios, a system that only accurately classifies 1% of images is useless even if it can do it in 0.001 seconds. A system that classifies 99% of images accurately is much preferred even if it takes almost a whole second to do so.

This section describes this relative level comparison methodology and Accuracy-to-Run Time ratio comparison to justify the choice of using run time as a metric secondary to classification accuracy. It is a poor choice to use both classification and run time as primary metrics because they do not properly represent the performance of a system.

As mentioned in the text of each factor experiment, the Wilcoxon signed-rank

test results in a p-value of 0.001 for every test except for the classification accuracy comparison between 3 rotations and 9 rotations. The Wilcoxon signed-rank test for these factor levels results in the $p = 0.556$. It may seem suspicious that every other test results in the exact same p-value that shows a significant difference between the two levels of each factor. However, this is a result of the small sample size of the test dataset. Although Wilcoxon signed-rank test does not directly rely on degrees of freedom, the principal of degrees of freedom applies similarly. Since the test only has 10 pieces of data to compare, it can only reach a certain granularity in the returned p-value. This granularity is $p = 0.001$ or more specifically $p = 0.001953125$ when $n = 10$. Therefore, the Wilcoxon signed-rank test reports a significant difference among almost all the levels of each factor with such confidence that it reports the lowest p-value possible. This test is proven to perform well on small datasets [101], and only $n = 6$ is required to test a hypothesis with a significance level of 0.05 [99]. Therefore, the use of $n = 10$ in experiment 3 tests all hypotheses confidently.

Putting statistical tests aside, the basic results of the Wilcox signed-rank can be derived with statistical intuition. At a glance, the means of the rotations' classification accuracies are within roughly 2 standard deviations of each other, so an insignificant value of $p = 0.556$ makes sense. However, every other comparison has two very different means both with small standard deviations. No other pairs of means appear to be within 10 standard deviations of each other, so it is easy to believe that they result in small p-values that indicate the factor levels are significant.

5.5 Results Summary

This chapter analyzes the results from experiments 1 and 2. These experiments prove both of the hypotheses which answer the first two of the three research questions. The third experiment answers the third research question: what factors most heavily

influence the identification of cellphone LiDAR light in a video? The third experiment identifies what factors influence the identification of cellphone LiDAR light in security camera video footage. It also analyzes potential values of these factors that maximize the classification accuracy and minimize the run time without decreasing the accuracy. These factor values are a 64 pixel SWOM window, 3 rotations, and 0 Lux light level. These factor values accurately classify LiDAR light in images 99.7% of the time in 91.978 seconds on average. This chapter concludes with a conceptual explanation of why these factors perform best in the system despite the fact that the chosen factor levels do not absolutely maximize accuracy and minimize run time.

VI. Conclusions and Recommendations

6.1 Overview

This chapter summarizes the research and experiment results. Section 6.2 addresses the research goals and hypotheses posed in Chapter I. Section 6.3 defines the contributions of this research followed by the limitations of this research in Section 6.4. Finally, Section 6.5 presents thoughts on future work, and Section 6.6 concludes with a summary of this research.

This research develops a framework to capture cellphone LiDAR light with a security camera, record it to a video, process the video frames to emphasize the cellphone LiDAR light pattern, and identify the presence of cellphone LiDAR light in these processed video frames with a neural network trained on a dataset generated from a single image of the cellphone LiDAR light pattern. This chapter elaborates on the level of success achieved with this framework as related to the defined research goals and hypotheses.

6.2 Research Conclusions

The main goal of this research is to detect cellphone LiDAR light with a security camera and positively identify it as LiDAR. This consists of multiple goals established with the following questions:

- Can a security camera reliably capture cellphone LiDAR light?
- Is there a reliable method to identify captured cellphone LiDAR light in a video?
- What factors most heavily influence the identification of cellphone LiDAR light in a video?

To address the first two of these goals, the following hypotheses are proposed:

1. In an environment with controlled textures and light levels, a security camera configuration exists that captures cellphone LiDAR light projected into the environment the majority of the time.

2. If video footage of cellphone LiDAR light is captured in an environment with controlled textures and light levels, each frame of the video can be processed to identify whether or not it contains cellphone LiDAR light with an accuracy above 50%.

The designed system tests these hypotheses with two main steps: record the cellphone LiDAR light with a security camera, and identify the LiDAR light in recorded footage. The security camera connects to a desktop workstation and is configured for use in recording video footage with the Python script. Identifying the LiDAR light in this footage consists of two steps: process the footage into training and test data, and build, train, and test a neural network to identify LiDAR light in images.

The system implements an effective Sliding Window Otsu's Method (SWOM) that isolates LiDAR light points in dark environments. A single image of these isolated LiDAR light points are used to simulate a training dataset of many 3D transformations of the LiDAR light as it may appear in video footage. To produce the test dataset, images of real LiDAR light are extracted from videos captured by the security camera. This saves images containing LiDAR light to use as positive samples in the test dataset and saves images that do not contain LiDAR light to use as negative samples in the test and training dataset.

The neural network is built according to [11] to classify patterns based on the frequency of pixel intensities in a 2D DFT. The neural network is trained on the training dataset which consists of simulated images that contain LiDAR light and real images that do not contain LiDAR light. The trained neural network is evaluated on the test dataset of real images that contain LiDAR light and real images that do not

contain LiDAR light. This neural network addresses part of the research goals and corresponding hypotheses.

Experiment 1 proves hypothesis 1 as detailed in Section 4.7.1. Experiment 2 proves hypothesis 2 as detailed in Section 4.7.2. Proving both of these hypotheses answers "Yes" to the first two research goals. As detailed in Section 4.7.3, experiment 3 then analyzes how and to what degree the SWOM window size, number of rotations, and light level affects the classification accuracy and run time in order to address the third research goal. The results of this experiment find that a 64 pixel SWOM window, 3 rotations, and 0 Lux light level perform best in the system. When trained on the 864 images of simulated LiDAR light produced by these factors as well as 450 real world images that do not contain LiDAR light, the neural network accurately classifies LiDAR light in images 99.7% of the time in 91.978 seconds on average.

6.3 Research Contributions

The process of capturing and recording LiDAR light may be novel as applied specifically to the bounds of this research, but the ability to capture LiDAR light with CMOS cameras is well known. However, this research does propose novel methodology to identify indirect ToF LiDAR light points in images. Beyond just LiDAR light points, this methodology can be used to identify other high contrast patterns as well as high contrast, disconnected objects. The ability to extract an object or pattern from the background of an image makes it much easier to identify. Even for purposes besides image classification, the Sliding Window Otsu's Method may prove useful in thresholding certain types of images. Furthermore, an image thresholded with this method may then be used as a mask to isolate the foreground pixels from the background in the original image. This may prove useful in other applications that require a robust background subtraction method for high contrast foreground objects.

Besides the image thresholding, this research also develops a method to project a 2D pattern into 3D space to simulate different ways that a camera might capture the pattern. Neural networks may use random flips and random rotations for training, but random affine transformations add much more overhead. The benefit of the neural network in this system is that the pattern classifier trains fast, so it is acceptable to spend more time generating data with affine transformations. Also, neural networks are not normally trained on the fly, so more time can be invested to generate a high quality dataset for the neural network to train on.

Although neural networks normally are not trained on the fly, this research develops a flexible system that facilitates creating datasets, building and training a neural network, and evaluating the neural network on real world images all in under 2 minutes. Collecting the two 30 second videos takes longer than the entire dataset generation and neural network training combined. The ability to train a neural network on a single image of a pattern and deploy it in under two minutes to identify other similar patterns with a 99%+ accuracy can support a wide range of technology. This is the best case scenario, but of course, many limitations currently prevent this system from seeing real world use.

6.4 Limitations of this Research

The largest limitation of this research is the simplicity of the environment. This research is specifically limited to a low light environment and a single flat wall upon which the LiDAR light projects. The trained neural network classifies images with high accuracy in a pitch black environment, but testing shows extremely diminished identification capabilities in even low light conditions. This research also assumes that the environment only contains Lambertian surfaces to ignore how LiDAR may reflect off a mirror or diffract through a window. This simplifies the design of the system,

but it also severely limits its application in a world full of windows and mirrors.

Another critical limitation is the required knowledge of the LiDAR light pattern. In a controlled research environment, it is not difficult to project LiDAR light from the iPhone 12 Pro Max and record its pattern in order to generate the training dataset. However, real world security cameras do not often point at flat walls in pitch black rooms where they can easily capture a clear image of a cellphone's LiDAR light pattern. Furthermore, the neural network is only trained on a single cellphone LiDAR pattern, so it can likely only identify LiDAR emitted from that cellphone. It is unrealistic to acquire all cellphones with LiDAR sensors and capture their LiDAR light patterns to generate a training dataset.

6.5 Recommendations for Future Work

This research develops a successful framework for detecting and identifying cellphone LiDAR light in a completely dark room, but future work to expand its capabilities can improve a few different parts.

- Contrast Amplification: The poor performance in brighter environments may be caused by the decreased contrast of the LiDAR light points. The lower the contrast of the LiDAR light points, the less accurately the SWOM operation isolates them. This could potentially be solved with a local or global contrast amplification operation before the SWOM operation.

- Distance Variation: The analysis of performance at different distances could better define the capabilities and limitations of the methodology developed in this research. This includes the distance of the camera to the wall, the distance of the cellphone to the wall, and potentially different angles of both devices.

- LiDAR Sensor Variation: Different 3D LiDAR sensors may have different light

points patterns. Evaluating how well this methodology identifies other LiDAR light patterns may lead to a general form solution that does not require prior knowledge of the LiDAR light pattern to identify it in an image.

- Kernel Size: The kernel size of the Gaussian blur, closing, and dilating operations in SWOM. This kernel controls how well the LiDAR light points are isolated. The size of the light points is directly affected by the distance of the phone and camera to the wall, so this also is likely influenced by the device distances and positions.

- Pattern Light Picture Isolation: The Preprocess_Footage.py script uses Otsu's Method to isolate the light points in the LiDAR light pattern image which it then uses to create the training dataset. This performs well in a 0 Lux environment but not a 35 Lux environment. The use of SWOM better isolates the light points in a 35 Lux environment, but a new method needs to be developed to automatically identify the corners and normalize the pattern to a square with an affine transformation.

6.6 Concluding Thoughts

In this research, a system is developed to detect and identify cellphone LiDAR light patterns in security camera video footage. The system performs well primarily in dark conditions, but it is a unique proof-of-concept of background removal with image thresholding to improve the pattern classification capabilities of a rapidly trained neural network.

Appendix A. Capture_Footage.py

This appendix contains the Python code for capturing and recording video footage from a connected Lorex security camera. Before executing this script, all items marked with arrows <like this> must be filled in. After executing the script, pressing the 'v' key on the keyboard records a 30 second long video at 15 FPS, and pressing the 'q' key will end the script. This code is adapted from [97].

```
import time
import datetime
import os
from threading import Thread
import cv2

dirOut = "<path>"
fps = 15
cameraIP = "<IP address>"
video_length = 30
user = '<username>'
pwd = '<password>'


class RTSPVideoWriterObject(object):
    def __init__(self, src):
        # Create a VideoCapture object
        self.capture = cv2.VideoCapture(src)

        # Default resolutions of the frame are obtained (system dependent)
        self.frame_width = int(self.capture.get(3))
        self.frame_height = int(self.capture.get(4))
```

```
23          self.last_cap = time.time()
24          self.write = 0
25
26          # Start the thread to read frames from the video stream
27          self.thread = Thread(target=self.update, args=())
28          self.thread.daemon = True
29          self.thread.start()
30
31      def update(self):
32          # Read the next frame from the stream in a different thread
33          while True:
34              if self.capture.isOpened():
35                  if(time.time() - self.last_cap >= 1/fps):
36                      self.last_cap = time.time()
37                      self.write = 1
38                      (self.status, self.frame) = self.capture.read()
39
40      def save_vid(self):
41          # Set up codec and output video settings
42          filename = f"{datetime.datetime.now():%Y-%m-%dT%H%M%S}" + '.avi'
43          file = os.path.join(dirOut, filename)
44          codec = cv2.VideoWriter_fourcc('M', 'J', 'P', 'G')
45          self.output_video = cv2.VideoWriter(file, codec, fps, (self.
    frame_width, self.frame_height))
46          # Save obtained frame into video output file
47          for i in range(video_length * fps):
48              while True:
49                  if self.write:
50                      self.output_video.write(self.frame)
51                      self.write = 0
52                      break
53          self.output_video.release()
```

```
54
55      def show_frame(self):
56          # Display frames in main program
57          if self.status:
58              cv2.imshow('frame', self.frame)
59
60          key = cv2.waitKey(1)
61          # Record video of length according to global variable
    video_length
62          # Pauses live footage while recording
63          if key == ord('v'):
64              self.save_vid()
65          # Press Q on keyboard to stop recording
66          elif key == ord('q'):
67              self.capture.release()
68              self.output_video.release()
69              cv2.destroyAllWindows()
70              exit(1)
71
72  if __name__ == '__main__':
73      # Connects with High Resolution 1920x1080
74      src = 'rtsp://' + user + ":" + pwd + "@" + cameraIP + ":554/cam/
    realmonitor?channel=1&subtype=0"
75      video_stream_widget = RTSPVideoWriterObject(src)
76      while True:
77          try:
78              video_stream_widget.show_frame()
79          except AttributeError:
80              pass
```

Appendix B. Preprocess_Footage.py

This appendix contains the Python code for generating the test and training datasets from videos captured from a security camera. When executed, the main function creates the datasets in the **path** folder with a SWOM window size of 64 pixels, 16 affine transformations, 3 rotations, and image size 512 x 512 pixels. Although the main function calls the **create_test_and_train_data()** function to generate the datasets, the Experiments.py script in Appendix E instead directly imports the **create_test_and_train_data()** function from this script to run the randomized experiments for each factor level.

```
import os
import cv2
import time
import math
import random
import imutils
import shutil
import numpy as np
import tensorflow as tf
from skimage.util import random_noise

from Layers import GetPatches

def mask_image(img):
    gray = img
    if (len(gray.shape) == 3):
        gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

    kernel = np.array([[0, 0, 1, 0, 0],
```

```
20                          [0, 1, 1, 1, 0],
21                          [1, 1, 1, 1, 1],
22                          [0, 1, 1, 1, 0],
23                          [0, 0, 1, 0, 0]], dtype=np.uint8)
24      gray = cv2.GaussianBlur(gray, (5, 5), 0)
25      gray = cv2.morphologyEx(gray, cv2.MORPH_CLOSE, kernel)
26      gray = cv2.dilate(gray, kernel, iterations=1)
27      ret, mask = cv2.threshold(gray, 0, 255, cv2.THRESH_BINARY + cv2.
    THRESH_OTSU)
28      return mask
29
30  def sliding_window_mask(img, win_size=64):
31      move_speed = win_size
32      x1 = 0
33      y1 = 0
34      x2 = win_size
35      y2 = win_size
36      vert_steps = math.ceil((len(img)-win_size)/move_speed)
37      hor_steps = math.ceil((len(img[0])-win_size)/move_speed)
38      for y in range(vert_steps + 1):
39          for x in range(hor_steps + 1):
40              sub_img = img[y1:y2, x1:x2]
41              masked = mask_image(sub_img)
42              img[y1:y2, x1:x2] = masked
43              x1 = x1 + move_speed
44              x2 = x2 + move_speed
45              if x2 > len(img[0]) - 1:
46                  x2 = len(img[0]) - 1
47                  x1 = x2 - win_size
48          x1 = 0
49          x2 = win_size
50          y1 = y1 + move_speed
```

```
51          y2 = y2 + move_speed
52          if y2 > len(img) - 1:
53              y2 = len(img) - 1
54              y1 = y2 - win_size
55      return img
56
57  def sliding_window(img, win_size):
58      move_speed = 100
59      y1 = 0
60      y2 = win_size
61      x1 = 0
62      x2 = win_size
63      images = []
64      for y in range(math.ceil((len(img)-win_size)/move_speed)):
65          for x in range(math.ceil((len(img[0])-win_size)/move_speed)):
66              sub_img = img[y1:y2, x1:x2]
67              images.append(sub_img)
68              x1 = x1 + move_speed
69              x2 = x2 + move_speed
70              if x2 > len(img[0]) - 1:
71                  x2 = len(img[0]) - 1
72                  x1 = x2 - win_size
73          x1 = 0
74          x2 = win_size
75          y1 = y1 + move_speed
76          y2 = y2 + move_speed
77          if y2 > len(img) - 1:
78              y2 = len(img) - 1
79              y1 = y2 - win_size
80      return np.array(images)
81
82  def get_bbox(img):
```

```
83      gray = img
84      if (len(img.shape) == 3):
85          gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
86      contours, _ = cv2.findContours(gray, cv2.RETR_EXTERNAL, cv2.
        CHAIN_APPROX_SIMPLE)
87      join_cnts = np.concatenate(contours)
88      return cv2.boundingRect(join_cnts)
89
90  def get_corners(img):
91      og_x, og_y, og_w, og_h = get_bbox(img)
92      cropped = img[og_y:(og_y + og_h), og_x:(og_x + og_w)]
93
94      split_multiplier = 4
95      split_height = int(len(cropped) / split_multiplier)
96      split_width = int(len(cropped[0]) / split_multiplier)
97      bot_split = split_height * (split_multiplier - 1)
98      right_split = split_width * (split_multiplier - 1)
99
100     top_left = cropped[:split_height, :split_width]
101     top_right = cropped[:split_height, right_split:]
102     bot_left = cropped[bot_split:, :split_width]
103     bot_right = cropped[bot_split:, right_split:]
104
105     x, y, w, h = get_bbox(top_left)
106     tl = [x, y]
107     x, y, w, h = get_bbox(top_right)
108     tr = [x + w + right_split, y]
109     x, y, w, h = get_bbox(bot_left)
110     bl = [x, y + h + bot_split]
111     x, y, w, h = get_bbox(bot_right)
112     br = [x + w + right_split, y + h + bot_split]
113     cropped_coords = [tl, tr, br, bl]
```

```
114     true_coords = np.array([[i + og_x, j + og_y] for [i, j] in
    cropped_coords], dtype=np.float32)
115     return true_coords
116
117 def normalize_image(img):
118     gray = img
119     if (len(img.shape) == 3):
120         gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
121
122     kernel = np.array([[0, 0, 1, 0, 0],
123                        [0, 1, 1, 1, 0],
124                        [1, 1, 1, 1, 1],
125                        [0, 1, 1, 1, 0],
126                        [0, 0, 1, 0, 0]], dtype=np.uint8)
127     gray = cv2.GaussianBlur(gray, (5, 5), 0)
128     gray = cv2.morphologyEx(gray, cv2.MORPH_CLOSE, kernel)
129     ret, mask = cv2.threshold(gray, 0, 255, cv2.THRESH_BINARY + cv2.
    THRESH_OTSU)
130
131     true_coords = get_corners(mask)
132     desired_coords = np.array([[680, 252], [1256, 252], [1256, 828],
    [680, 828]], dtype=np.float32)
133     M = cv2.getPerspectiveTransform(true_coords, desired_coords)
134     transformed_gray = cv2.warpPerspective(gray, M, (len(gray[0]), len(
    gray)))
135     transformed_mask = cv2.warpPerspective(mask, M, (len(gray[0]), len(
    gray)))
136     masked = cv2.bitwise_and(transformed_gray, transformed_mask)
137     return masked[252:828, 680:1256]
138
139 def affine_transform(img, i, j, split):
```

```
140     true_coords = np.array([[0, 0], [len(img[0]) - 1, 0], [len(img[0]) -
        1, len(img) - 1], [0, len(img) - 1]], dtype=np.float32)
141     tl = [i * split, j * split]
142     tr = true_coords[1]
143     br = true_coords[2]
144     bl = [i * split, (j + 1) * split]
145     desired_coords = np.array([tl, tr, br, bl], dtype=np.float32)
146     M = cv2.getPerspectiveTransform(true_coords, desired_coords)
147     left_dst = cv2.warpPerspective(img, M, (len(img[0]), len(img)))
148     #left_dst = cv2.cvtColor(dst, cv2.COLOR_BGR2GRAY)
149
150     tl = [j * split, i * split]
151     tr = [(j + 1) * split, i * split]
152     br = true_coords[2]
153     bl = true_coords[3]
154     desired_coords = np.array([tl, tr, br, bl], dtype=np.float32)
155     M = cv2.getPerspectiveTransform(true_coords, desired_coords)
156     top_dst = cv2.warpPerspective(img, M, (len(img[0]), len(img)))
157     #top_dst = cv2.cvtColor(dst, cv2.COLOR_BGR2GRAY)
158     return left_dst, top_dst
159
160 def affine_transforms(img, num_transforms):
161     # LiDAR pattern is only symmetrically diagonally
162     # Make 90 degree rotate copy to cover all permutations
163     transformed_imgs = []
164     rot_img = imutils.rotate(img, 90)
165     transformed_imgs.append(img)
166     transformed_imgs.append(rot_img)
167
168     # Transform num_transforms different ways
169     if num_transforms > 1:
170         split = int(len(img) / np.sqrt(num_transforms))
```

```
171         for i in range(int(np.sqrt(num_transforms))):
172             for j in range(int(np.sqrt(num_transforms))):
173                 left_dst, top_dst = affine_transform(img, i, j, split)
174                 transformed_imgs.append(left_dst)
175                 transformed_imgs.append(top_dst)
176     return transformed_imgs
177
178 def get_patches(imgs, img_size):
179     imgs = np.expand_dims(imgs, -1)
180     patches4 = GetPatches(num_patches=4)(imgs)
181     patches4 = tf.reshape(patches4, [-1, patches4.shape[-3], patches4.
      shape[-2], 1])
182     patches16 = GetPatches(num_patches=4)(patches4)
183     patches16 = tf.reshape(patches16, [-1, patches16.shape[-3],
      patches16.shape[-2], 1])
184     imgs = np.squeeze(imgs, axis=-1)
185
186     patches = []
187     for patch in patches4.numpy():
188         res = cv2.resize(patch, (img_size, img_size))
189         patches.append(res)
190     for patch in patches16.numpy():
191         res = cv2.resize(patch, (img_size, img_size))
192         patches.append(res)
193     patches = np.array(patches)
194     patches = np.concatenate((imgs, patches), axis=0)
195     return patches
196
197 def validate_patches(patches):
198     valid_patches = []
199     for patch in patches:
```

```
200         ret, mask = cv2.threshold(patch, 0, 255, cv2.THRESH_BINARY + cv2
    .THRESH_OTSU)
201         contours, _ = cv2.findContours(mask, cv2.RETR_EXTERNAL, cv2.
    CHAIN_APPROX_SIMPLE)
202         num_dots = len(contours)
203         if num_dots > 16:
204             join_cnts = np.concatenate(contours)
205             x, y, w, h = cv2.boundingRect(join_cnts)
206             bbox_area = w * h
207             total_area = len(patch) * len(patch[0])
208             coverage = bbox_area / total_area
209             if coverage > 0.50:
210                 valid_patches.append(patch)
211
212     return np.array(valid_patches)
213
214 def permute(img, path, num_transforms, num_rotations, win_size, img_size
    ):
215     transformed_imgs = affine_transforms(img, num_transforms)
216     patches = get_patches(transformed_imgs, img_size)
217     valid_patches = validate_patches(patches)
218
219     write_count = 0
220     for patch in valid_patches:
221         rotations = []
222         for r in range(num_rotations):
223             rotated = imutils.rotate(patch, random.randint(0, 360))
224             rotations.append(rotated)
225         for pic in rotations:
226             noisy = random_noise(pic, mode='gaussian', var=0.05 ** 2)
227             noisy = (255 * noisy).astype(np.uint8)
228             mask = sliding_window_mask(noisy, win_size)
```

```
229             cv2.imwrite(path + 'lidar\\' + str(write_count) + '.png',
    mask)
230             write_count += 1
231
232 def create_train_dataset(path, num_transforms=9, num_rotations=3,
    win_size=64, img_size=512):
233     train_data_path = path + "\\train\\"
234     # Creates folders recursively if they do not exist:
235     # path\train\
236     # path\train\lidar\
237     # path\train\neg\
238     os.makedirs(train_data_path + 'lidar', exist_ok=True)
239     #os.makedirs(train_data_path + 'neg', exist_ok=True)
240     dst = path+'\\train\\neg'
241     try:
242         shutil.rmtree(dst)
243     except OSError as e:
244         pass
245     if 'light' in path:
246         shutil.copytree(path + '\\test\\masked_pics\\light\\neg', dst)
247     else:
248         shutil.copytree(path + '\\test\\masked_pics\\dark\\neg', dst)
249
250     pic = path + '\\light_pattern.png'
251     img = cv2.imread(pic, cv2.IMREAD_GRAYSCALE)
252     normalized = normalize_image(img)
253     normalized = cv2.resize(normalized, (img_size, img_size))
254     permute(normalized, train_data_path, num_transforms, num_rotations,
    win_size, img_size)
255
256 def create_pic_directories(video_path):
257     # Split up video path by slashes
```

```
258     split_path = video_path.split('\\')
259     # Get video name from end of split list and remove .avi extension
260     vid_name = split_path[-1].split('.')[0]
261     # Remove top folder and join to get path to parent directory
262     parent_path = ('\\').join(split_path[:-2])
263     # Remove '_neg' from file name so we direct to the same pic folders
        as the lidar video
264     if '_neg' in vid_name:
265         vid_name = vid_name.replace('_neg', '')
266     # Create path of folder in raw_pics that will contain pics for this
        video
267     raw_pics_path = parent_path + '\\raw_pics\\' + vid_name + '\\'
268     masked_pics_path = parent_path + '\\masked_pics\\' + vid_name + '\\'
269
270     # Creates folders recursively if they do not exist:
271     # path\test\raw_pics
272     # path\test\raw_pics\<vid_name>
273     # path\test\raw_pics\<vid_name>\lidar
274     # path\test\raw_pics\<vid_name>\neg
275     os.makedirs(raw_pics_path + 'lidar', exist_ok=True)
276     os.makedirs(raw_pics_path + 'neg', exist_ok=True)
277
278     # We do not use these immediately in the caller function
279     # but it is most convenient to create the folders now
280     # Creates folders recursively if they do not exist:
281     # path\test\masked_pics
282     # path\test\masked_pics\<vid_name>
283     # path\test\masked_pics\<vid_name>\lidar
284     # path\test\masked_pics\<vid_name>\neg
285     os.makedirs(masked_pics_path + 'lidar', exist_ok=True)
286     os.makedirs(masked_pics_path + 'neg', exist_ok=True)
287
```

```
288     return raw_pics_path
289
290 def vid_to_frames(video_path, img_size):
291     # Create necessary folders
292     # Returns path to created lidar folder
293     raw_pics_path = create_pic_directories(video_path)
294     lidar_path = raw_pics_path + 'lidar'
295     neg_path = raw_pics_path + 'neg'
296
297
298     # Get video name from video_path
299     split_path = video_path.split('\\')
300     # Remove .avi extension from video name
301     vid_name = split_path[-1].split('.')[0]
302
303     img_count = 0
304     cap = cv2.VideoCapture(video_path)
305     ret, og_img = cap.read()
306
307     root = video_path.split('\\')[0]
308     pattern_path = root + '\\light_pattern.png'
309     if not os.path.isfile(pattern_path) and 'neg' not in vid_name:
310         cv2.imwrite(pattern_path, og_img)
311
312     while og_img is not None:
313         # Get full path to store a single frame it is respective \
    raw_pics\<video>\lidar folder
314         frame_path = os.path.join(lidar_path, vid_name + "_" + str(
    img_count) + ".png")
315
316         # Store videos with 'neg' in the title to the respective neg
    folder instead of the lidar folder
```

```
317         #\raw_pics\<video>\neg
318         if 'neg' in vid_name:
319             frame_path = os.path.join(neg_path, vid_name + "_" + str(
    img_count) + ".png")
320 
321         # Extract middle square that contains majority of lidar light
    points
322         crop = og_img[65:1015, 500:1450]
323         res = cv2.resize(crop, (img_size, img_size))
324         cv2.imwrite(frame_path, res)
325         img_count += 1
326         ret, og_img = cap.read()
327 
328 def all_vids_to_frames(path, img_size):
329     vid_path = path + '\\videos'
330     videos = []
331     for file in os.listdir(vid_path):
332         if '.avi' in file:
333             videos.append(os.path.join(vid_path, file))
334     for video in videos:
335         vid_to_frames(video, img_size)
336 
337 def convert_raw_pics_to_masked_pics(dataset_path, win_size):
338     raw_pic_paths = []
339     for subdir, dirs, files in os.walk(dataset_path + '\\raw_pics'):
340         for file in files:
341             raw_pic_paths.append(os.path.join(subdir, file))
342     for raw_pic_path in raw_pic_paths:
343         raw_pic = cv2.imread(raw_pic_path, cv2.IMREAD_GRAYSCALE)
344         masked_img = sliding_window_mask(raw_pic, win_size)
345         masked_pic_path = raw_pic_path.replace('raw_pics', 'masked_pics'
    )
```

```
346             cv2.imwrite(masked_pic_path, masked_img)
347
348 def create_test_dataset(dataset_path, win_size=64, img_size=512):
349     test_data_path = dataset_path + "\\test"
350     if not os.path.isdir(test_data_path + '\\raw_pics'):
351         # Create folder for each video
352         # Save all frames as images to respective folder
353         all_vids_to_frames(test_data_path, img_size)
354     # Read all the images above in raw_pics
355     # Mask them and save them to <dataset_path>\masked_pics\
356     convert_raw_pics_to_masked_pics(test_data_path, win_size)
357
358 def create_test_and_train_data(path, win_size, num_transforms,
    num_rotations, img_size):
359     # In same directory as script, create file structure
360     # <path>\
361     # <path>\light_pattern.png (LiDAR light pattern image to generate
    training images)
362     # <path>\test\
363     # <path>\test\videos (containing test videos)
364     create_test_dataset(path, win_size=win_size, img_size=img_size)
365     create_train_dataset(path, num_transforms=num_transforms,
    num_rotations=num_rotations, win_size=win_size, img_size=img_size)
366
367 if __name__ == '__main__':
368     start_time = time.time()
369     create_test_and_train_data('path', win_size=64, num_transforms=16,
    num_rotations=3, img_size=512)
370     print("%f seconds" % (time.time() - start_time))
```

Appendix C. Layers.py

This appendix contains the Python code used primarily for building the neural network. The **GetPatches** class splits images into the provided number of patches. The **GetBatchCosts** class sums the pixel intensities in an image based on what pooling ring they fall into. The corresponding pooling ring for each pixel is decided by its Chebyshev distance from the center of the image. The **ApplyWeights** class multiplies the pixel intensity sums from GetBatchCosts by trainable weights that SGD updates. SGD is able to automatically find the gradient of these classes since they are implemented exclusively in TensorFlow functions. These three classes implement the frequency pattern classifier described in [11].

```
1  import tensorflow as tf
2
3  class GetPatches( tf.keras.layers.Layer ):
4      def __init__(self, num_patches):
5          super(GetPatches, self).__init__()
6          self.num_patches = tf.constant(num_patches, dtype=tf.float32)
7
8      def build(self, input_shape):
9          self.batch_size = input_shape[0]
10         self.patch_size = tf.cast(tf.math.divide(input_shape[1], tf.math
       .sqrt(self.num_patches)), dtype=tf.int32)
11         #self.total_num_patches = tf.cast(tf.multiply(input_shape[0],
       self.num_patches), dtype=tf.int32)
12
13     def call(self, inputs):
14         sizes = [1, self.patch_size, self.patch_size, 1]
15         patches = tf.image.extract_patches(images=inputs,
16                                            sizes=sizes,
```

```
17                                      strides=sizes,
18                                      rates=[1, 1, 1, 1],
19                                      padding='VALID')
20         return tf.reshape(patches, [self.batch_size, tf.cast(self.
    num_patches, dtype=tf.int32), self.patch_size, self.patch_size, 1])
21
22 class GetBatchCosts(tf.keras.layers.Layer):
23     def __init__(self, ring_size=1, patches=False):
24         super(GetBatchCosts, self).__init__()
25         self.ring_size = tf.constant(ring_size, dtype=tf.float64)
26         self.patches = patches
27
28     def sum_rings(self, inputs):
29         inputs_shape = tf.shape(inputs)
30         inputs = tf.reshape(inputs, [inputs_shape[0], inputs_shape[1],
    inputs_shape[2]])
31         inputs_shape = tf.shape(inputs)
32
33         # input_shape.shape[1] == input_shape.shape[2]
34         tf.assert_equal(inputs_shape[1], inputs_shape[2], message='
    Cannot create rings for non-square fourier image.')
35
36         # math.ceil(input_shape.shape[1] / 2) % self.ring_size == 0
37         tf.assert_equal(tf.math.mod(tf.math.ceil(tf.math.truediv(
    inputs_shape[1], 2)), self.ring_size),
38                         tf.constant(0, dtype=tf.float64),
39                         message='Height of fourier image is not evenly
    divisible into rings')
40         # math.ceil(input_shape.shape[2] / 2) % self.ring_size == 0
41         tf.assert_equal(tf.math.mod(tf.math.ceil(tf.math.truediv(
    inputs_shape[2], 2)), self.ring_size),
42                         tf.constant(0, dtype=tf.float64),
```

```
43                          message='Width of fourier image is not evenly
    divisible into rings')
44
45          tf.assert_equal(inputs_shape[1], inputs_shape[2],
46                          message='Cannot create rings for non-square
    fourier image.')
47
48          # The sparse matrix will not track zero values returned by the
    fourier transform
49          # so we add a small value to each element after the fourier
    transform
50          tf_fft = tf.abs(tf.signal.fftshift(tf.signal.fft2d(tf.cast(
    inputs, dtype=tf.complex64)))) + tf.constant(0.001)
51          tf_fft_shape = tf.shape(tf_fft)
52          ef = tf.reshape(tf_fft, [tf_fft_shape[0], tf_fft_shape[1],
    tf_fft_shape[2]])
53          st = tf.sparse.from_dense(ef)
54
55          # Transform sparse matrix indices into batches format of (
    batch_num, image, ef_index)
56          st_indices = tf.cast(tf.identity(st.indices), dtype=tf.float32)
57          batch_nums, ef_idxs = tf.split(st_indices, [1, 2], axis=1)
58          batched_ef_idxs = tf.reshape(ef_idxs, [inputs_shape[0], -1, 2])
59
60          # Calculate Chebyshev distance of each pixel from the center
    that is (mid, mid)
61          ef_shape = tf.shape(ef)
62          mid_val = tf.divide(tf.subtract(ef_shape[-2], 1), 2)
63          mid = tf.cast(tf.repeat(mid_val, repeats=[2]), dtype=tf.float32)
64          # Only calculate distances for first image since all images are
    same size
65          first_batched_ef_idxs = tf.expand_dims(batched_ef_idxs[0], 0)
```

```
66          sub = tf.subtract(first_batched_ef_idxs, mid)
67          abs_t = tf.abs(sub)
68          maxxed = tf.reduce_max(abs_t, axis=2)
69          floored = tf.floor(maxxed)
70          floored = tf.cast(floored, dtype=tf.float64)
71          div = tf.divide(floored, self.ring_size)
72          dist = tf.cast(div, dtype=tf.int32)
73
74          # Sum the pixel values of the fourier transformed image based on
        distance from center
75          num_rings = tf.cast(tf.divide(tf.math.ceil(tf.divide(ef_shape
    [-2], 2)), self.ring_size), dtype=tf.int32)
76          res_inputs = tf.transpose(tf.reshape(ef, [ef_shape[0], -1]))
77
78          summed_rings = tf.transpose(tf.math.unsorted_segment_sum(
    res_inputs, dist[0], num_rings))
79          return summed_rings
80
81      def call(self, inputs):
82          if self.patches:
83              inputs_shape = tf.shape(inputs)
84              return tf.reshape(tf.map_fn(self.sum_rings, inputs), [
    inputs_shape[0], -1])
85          else:
86              return self.sum_rings(inputs)
87
88  class ApplyWeights(tf.keras.layers.Layer):
89      def __init__(self):
90          super(ApplyWeights, self).__init__()
91
92      def build(self, input_shape):
93          self.w = self.add_weight(
```

```
94              shape=(input_shape[1], ),
95              initializer=tf.keras.initializers.RandomUniform(minval=0,
    maxval=0.3),
96              trainable=True,
97              constraint=tf.keras.constraints.NonNeg(),
98              name='ring_weights'
99          )
100
101     def call(self, inputs):
102         return tf.multiply(inputs, self.w)
```

Appendix D. Train_Evaluate_NN.py

This appendix contains the Python code that builds a neural network from [11] then trains and evaluates it on the training and test datasets respectively. The **get_model()** function builds the model out of generic TensorFlow layers as well as layers imported from the Layers.py script detailed in Appendix C. The **get_dataset()** function loads either the test or the training dataset created by the Preprocess_Footage.py script detailed in Appendix B. The **train_and_save_model()** function builds the model, trains it on the training dataset, then saves the trained model with all of its trained variable values to disk. This allows to the model to be loaded and used regardless of access to the source code. The **load_and_evaluate_model()** function loads the saved model and returns the accuracy of the trained neural network as evaluated on the test dataset. Although the main function calls **train_and_save_model()** and **load_and_evaluate_model()** to build, train, and test the dataset, the Experiments.py script in Appendix E instead directly imports these functions to run the randomized experiments for each factor level.

```
1  import os
2  import time
3  import random
4  import pathlib
5  import shutil
6  import tensorflow as tf
7  from Layers import GetPatches, GetBatchCosts, ApplyWeights
8
9  def get_model(img_size, batch_size, ring_size):
10     inputs = tf.keras.Input(shape=(img_size, img_size, 1), batch_size=
       batch_size)
11     patches_4 = GetPatches(num_patches=4)(inputs)
```

```
12      patches_16 = GetPatches(num_patches=16)(inputs)
13      costs1 = GetBatchCosts(ring_size=ring_size)(inputs)
14      costs4 = GetBatchCosts(ring_size=ring_size, patches=True)(patches_4)
15      costs16 = GetBatchCosts(ring_size=ring_size, patches=True)(
        patches_16)
16      costs = tf.keras.layers.concatenate([costs1, costs4, costs16], axis
        =1)
17      weighted_costs = ApplyWeights()(costs)
18      dense1 = tf.keras.layers.Dense(512, activation='relu', trainable=
        True)(weighted_costs)
19      sigmoid = tf.keras.layers.Dense(1, activation='sigmoid')(dense1)
20      model = tf.keras.Model(inputs=inputs, outputs=sigmoid)
21      tf.keras.utils.plot_model(model, to_file='model.png', show_shapes=
        True)
22      return model
23
24  def get_dataset(pics_path, img_size, batch_size):
25      data_dir = pathlib.Path(pics_path)
26      train_ds = tf.keras.utils.image_dataset_from_directory(data_dir,
27          validation_split=0.2,
28          shuffle=True,
29          subset='training',
30          color_mode='grayscale',
31          seed=random.randint(0, 10000),
32          image_size=(img_size, img_size),
33          batch_size=batch_size,
34          label_mode='binary')
35      val_ds = tf.keras.utils.image_dataset_from_directory(data_dir,
36           validation_split=0.2,
37           subset='validation',
38           shuffle=True,
39           color_mode='grayscale',
```

```
40          seed=random.randint(0, 10000),
41          image_size=(img_size, img_size),
42          batch_size=batch_size,
43          label_mode='binary')
44      return train_ds, val_ds
45
46  def train_and_save_model(path, img_size=512):
47      batch_size = 1
48      ring_size = 1
49      model = get_model(img_size, batch_size, ring_size)
50      train_ds, val_ds = get_dataset(path + '\\train', img_size,
        batch_size)
51
52      model.compile(
53          optimizer=tf.keras.optimizers.Adam(learning_rate=0.00001),
54          loss=tf.keras.losses.BinaryCrossentropy(),
55          metrics=[tf.keras.metrics.BinaryAccuracy()]
56      )
57
58      callback = tf.keras.callbacks.EarlyStopping(monitor='loss', patience
        =3)
59      model.fit(train_ds, validation_data=val_ds, batch_size=batch_size,
        epochs=5, callbacks=[callback])
60
61      # Delete old model if it exists
62      dirpath = pathlib.Path(path + '\\model')
63      if dirpath.exists() and dirpath.is_dir():
64          shutil.rmtree(dirpath)
65      # Save new model
66      model.save(path + '\\model')
67
68  def load_and_evaluate_model(path, img_size=512):
```

```
69      batch_size = 1
70      tests = path + '\\test\\masked_pics'
71      test_paths = []
72      test_names = []
73      for test in os.listdir(tests):
74          test_paths.append(os.path.join(tests, test))
75          test_names.append(test)
76
77      model = tf.keras.models.load_model(path + '\\model')
78      results = []
79      for test_path, test_name in zip(test_paths, test_names):
80          dataset = tf.keras.utils.image_dataset_from_directory(test_path,
81              color_mode='grayscale',
82              shuffle=True,
83              image_size=(img_size, img_size),
84              batch_size=batch_size,
85              label_mode='binary')
86
87          accuracy = model.evaluate(dataset, verbose=0)
88          results.append(test_name + ' - Accuracy: ' + str(accuracy[1]))
89
90      return results
91
92
93  if __name__ == '__main__':
94      start_time = time.time()
95      train_and_save_model('path')
96      results = load_and_evaluate_model('path')
97      for result in results:
98          print(result)
99      print("{:.9f} seconds".format(time.time() - start_time))
```

Appendix E. Experiments.py

This appendix contains the Python code to run 10 randomized trials for each level of each factor in Experiment 3. Each experiment function is wrapped with many lines of logistical code for recording the accuracies and run times in a convenient format for use in the Statistical_Analysis.py Python script in Appendix F. However, the core of each experiment function uses **create_test_and_train_data()** from Preprocess_Footage.py to create the test and training datasets then uses **train_and_save_model()** and **load_and_evaluate_model()** from Train_Evaluate_NN.py to build, train, and test the neural network. After each training and test of the neural network, the **reset()** function erases everything in the training and test datasets folder except for the **test\videos** and **test\raw_pics** folder.

Each factor is tested in one function. The **window_sizes_32_vs_64()** function records the accuracies and run times for 20 randomized trials consisting of 10 trials for a 32 pixel SWOM window and 10 trials for a 64 pixel SWOM window. The number of rotations and light level are held constant at 3 and 0 Lux respectively. The **rotation_num_3_vs_9()** function records the accuracies and run times for 20 randomized trials consisting of 10 trials for 3 rotations and 10 trials for 9 rotations. The SWOM window size and light level are held constant at 64 pixels and 0 Lux respectively. The **dark_vs_light()** function records the accuracies and run times for 20 randomized trials consisting of 10 trials for a 0 Lux light level and 10 trials for a 35 Lux light level. The SWOM window size and number of rotations are held constant at 64 pixels and 3 respectively. The accuracies and run times from each function are saved to text files which are then copy and pasted in to Statistical_Analysis.py.

```
import os
import time
import shutil
import pathlib
import numpy as np

from Preprocess_Footage import create_test_and_train_data
from Train_Evaluate_NN import train_and_save_model,
    load_and_evaluate_model

# Delete everything in the environment except for
# <env>\test\raw_pics
# <env>\test\videos
def reset(env_path):
    model_path = pathlib.Path(env_path + '\\model')
    train_path = pathlib.Path(env_path + '\\train')
    paths = [model_path, train_path]
    for p in paths:
        if p.exists() and p.is_dir():
            shutil.rmtree(p)

    mp_path = env_path + '\\test\\masked_pics'
    if os.path.isdir(mp_path):
        for subdir, dirs, files in os.walk(mp_path):
            for file in files:
                os.remove(os.path.join(subdir, file))

def window_sizes_32_vs_64(env_path):
    win_sizes = np.array([32, 64]*10)
    np.random.shuffle(win_sizes)
    experiment_count = 0
    experiment_results = []
```

```
32      write_results = []
33      accuracy_32 = 'accuracy_32 = ['
34      accuracy_64 = 'accuracy_64 = ['
35      time_32 = 'time_32 = ['
36      time_64 = 'time_64 = ['
37      for win_size in win_sizes:
38          reset(env_path)
39          experiment_time = time.time()
40          img_size = 512
41          num_rotations = 3
42          create_test_and_train_data(env_path, win_size=win_size,
    num_transforms=16, num_rotations=num_rotations, img_size=img_size)
43          train_and_save_model(env_path, img_size)
44          results = load_and_evaluate_model(env_path, img_size)
45
46          elapsed_time = '{:.9f}'.format(time.time() - experiment_time)
47          header = 'Window Size = {} completed in {} seconds\n'.format(
    win_size, elapsed_time)
48          write_results.append(header)
49          toks = results[0].split(' ')
50          accuracy = toks[-1]
51          write_results.append(accuracy + '\n')
52          write_results.append('\n')
53
54          # Format accuracy and time nicely for use in Statistic_Analysis.
    py
55          if win_size==32:
56              accuracy_32 = accuracy_32 + accuracy + ', '
57              time_32 = time_32 + elapsed_time + ', '
58          elif win_size==64:
59              accuracy_64 = accuracy_64 + accuracy + ', '
60              time_64 = time_64 + elapsed_time + ', '
```

```
61          else:
62              print("ENCOUNTERED UNEXPECTED WINDOW SIZE {}".format(
    win_size))
63              exit()
64
65          experiment_count += 1
66          print('COMPLETED EXPERIMENT #{}'.format(experiment_count))
67          print(header)
68
69      write_results.append(accuracy_32[:-2] + ']\n')
70      write_results.append(accuracy_64[:-2] + ']\n')
71      write_results.append(time_32[:-2] + ']\n')
72      write_results.append(time_64[:-2] + ']\n')
73
74      for line in experiment_results:
75          print(line)
76
77      with open('results_winsize_32_vs_64.txt', 'w') as fd:
78          fd.writelines(write_results)
79
80  def rotation_num_3_vs_9(env_path):
81      rot_nums = np.array([3, 9]*10)
82      np.random.shuffle(rot_nums)
83
84      experiment_count = 0
85      experiment_results = []
86      write_results = []
87      accuracy_3 = 'accuracy_3 = ['
88      accuracy_9 = 'accuracy_9 = ['
89      time_3 = 'time_3 = ['
90      time_9 = 'time_9 = ['
91      for rot_num in rot_nums:
```

```
92          reset(env_path)
93          experiment_time = time.time()
94          img_size = 512
95          win_size = 64
96          create_test_and_train_data(env_path, win_size=win_size,
    num_transforms=16, num_rotations=rot_num, img_size=img_size)
97          train_and_save_model(env_path, img_size)
98          results = load_and_evaluate_model(env_path, img_size)
99
100         elapsed_time = '{:.9f}'.format(time.time() - experiment_time)
101         header = 'Rotation Number = {} completed in {} seconds\n'.format
    (rot_num, elapsed_time)
102         write_results.append(header)
103         toks = results[0].split(' ')
104         accuracy = toks[-1]
105         write_results.append(accuracy + '\n')
106         write_results.append('\n')
107
108         # Format accuracy and time nicely for use in Statistic_Analysis.
    py
109         if rot_num==3:
110             accuracy_3 = accuracy_3 + accuracy + ', '
111             time_3 = time_3 + elapsed_time + ', '
112         elif rot_num==9:
113             accuracy_9 = accuracy_9 + accuracy + ', '
114             time_9 = time_9 + elapsed_time + ', '
115         else:
116             print("ENCOUNTERED UNEXPECTED ROTATION NUMBER {}".format(
    win_size))
117             exit()
118
119         experiment_count += 1
```

```
120             print('COMPLETED EXPERIMENT #{}'.format(experiment_count))
121             print(header)
122 
123         write_results.append(accuracy_3[:-2] + ']\n')
124         write_results.append(accuracy_9[:-2] + ']\n')
125         write_results.append(time_3[:-2] + ']\n')
126         write_results.append(time_9[:-2] + ']\n')
127 
128         for line in experiment_results:
129             print(line)
130 
131         with open('results_rotnum_3_vs_9.txt', 'w') as fd:
132             fd.writelines(write_results)
133 
134 def dark_vs_light(env_path, light_env_path):
135         light_levels = np.array([env_path, light_env_path] * 10)
136         np.random.shuffle(light_levels)
137 
138         experiment_count = 0
139         experiment_results = []
140         write_results = []
141         accuracy_dark = 'accuracy_dark = ['
142         accuracy_light = 'accuracy_light = ['
143         time_dark = 'time_dark = ['
144         time_light = 'time_light = ['
145         for light_level in light_levels:
146             reset(light_level)
147             experiment_time = time.time()
148             img_size = 512
149             win_size = 64
150             rot_num = 3
151             create_test_and_train_data(light_level, win_size=win_size,
```

```
num_transforms=16, num_rotations=rot_num, img_size=img_size)
152         train_and_save_model(light_level, img_size)
153         results = load_and_evaluate_model(light_level, img_size)
154 
155         elapsed_time = '{:.9f}'.format(time.time() - experiment_time)
156         header = 'Light Level Path = {} completed in {} seconds\n'.
    format(light_level, elapsed_time)
157         write_results.append(header)
158         toks = results[0].split(' ')
159         accuracy = toks[-1]
160         write_results.append(accuracy + '\n')
161         write_results.append('\n')
162 
163         # Format accuracy and time nicely for use in Statistic_Analysis.
    py
164         if 'light' in light_level:
165             accuracy_light = accuracy_light + accuracy + ', '
166             time_light = time_light + elapsed_time + ', '
167         else:
168             accuracy_dark = accuracy_dark + accuracy + ', '
169             time_dark = time_dark + elapsed_time + ', '
170 
171         experiment_count += 1
172         print('COMPLETED EXPERIMENT #{}'.format(experiment_count))
173         print(header)
174 
175     write_results.append(accuracy_dark[:-2] + ']\n')
176     write_results.append(accuracy_light[:-2] + ']\n')
177     write_results.append(time_dark[:-2] + ']\n')
178     write_results.append(time_light[:-2] + ']\n')
179 
180     for line in experiment_results:
```

```
181         print(line)
182
183     with open('results_lightlevel_dark_vs_light.txt', 'w') as fd:
184         fd.writelines(write_results)
185
186 if __name__ == '__main__':
187     start_time = time.time()
188     window_sizes_32_vs_64('path')
189     rotation_num_3_vs_9('path')
190     dark_vs_light('path', 'path_light')
191     print("All experiments completed in {:.9f} seconds".format(time.time
        () - start_time))
```

Appendix F. Statistic_Analysis.py

This appendix contains the Python code to calculate the mean and standard deviation of collections of classification accuracies and run times from the Experiments.py script in Appendix E. It also uses the **stats.wilcoxon()** function from SciPy version 1.7.1 to identify significant differences in the run times and accuracies produced by the two levels of a single factor. Lines 24-37 are copy and pasted from the text files saved by Experiments.py. Lines 39-41 call the **compare_distributions()** function with the two accuracy arrays and two run time arrays corresponding to the two levels of a single factor. This **compare_distributions()** function calculates the mean and standard deviation of each accuracy and run time array as well as the Wilcoxon signed-rank test of the accuracy and test arrays. This results in a total of 10 values for each factor: 4 means, 4 standard deviations, and 2 Wilcoxon signed-rank test results. These values are analyzed in Section 5.3.

```
1  import scipy.stats as stats
2  import numpy as np
3
4  def compare_distributions(accuracyA, accuracyB, timeA, timeB):
5      if accuracyA != None and accuracyB != None:
6          # Convert to numpy arrays so we can use .mean()
7          accuracyA = np.array(accuracyA)
8          accuracyB = np.array(accuracyB)
9          _, accuracy_p = stats.wilcoxon(accuracyA, accuracyB, correction=
       False)
10         print("Significance of Accuracy: {}".format(accuracy_p))
11         print("Accuracy A - Mean = {:.16f}, STD = {:.16f}".format(
       accuracyA.mean(), accuracyA.std()))
12         print("Accuracy B - Mean = {:.16f}, STD = {:.16f}".format(
       accuracyB.mean(), accuracyB.std()))
```

```
13
14      if timeA != None and timeB != None:
15          timeA = np.array(timeA)
16          timeB = np.array(timeB)
17          _, time_p = stats.wilcoxon(timeA, timeB, correction=False)
18          print("Significance of Computation Time: {}".format(time_p))
19          print("Time A - Mean = {:.9f}, STD = {:.9f}".format(timeA.mean()
    , timeA.std()))
20          print("Time B - Mean = {:.9f}, STD = {:.9f}".format(timeB.mean()
    , timeB.std()))
21          print()
22
23  accuracy_32 = [0.9522222280502319, 0.9166666865348816,
        0.8744444251060486, 0.9266666769981384, 0.9388889074325562,
        0.9055555462837219, 0.9044444561004639, 0.8677777647972107,
        0.852222204208374, 0.9066666960716248]
24  accuracy_64 = [0.995555579662323, 1.0, 1.0, 0.9866666793823242,
        0.9911110997200012, 1.0, 1.0, 0.995555579662323, 1.0,
        0.996666669845581]
25  time_32 = [111.682993650, 109.717566490, 109.409659386, 110.412479877,
        110.952991724, 113.152074099, 108.855632305, 109.632296324,
        112.794996977, 111.738496065]
26  time_64 = [91.070557833, 90.648389816, 90.882352352, 95.847110987,
        92.504647970, 93.496391535, 90.497029781, 90.682560682,
        92.726378918, 91.427412987]
27
28  accuracy_3 = [0.9933333396911621, 0.996666669845581, 0.996666669845581,
        1.0, 0.9811111092567444, 0.995555579662323, 0.9866666793823242,
        0.996666669845581, 0.9866666793823242, 0.9833333492279053]
29  accuracy_9 = [0.9744444489479065, 1.0, 1.0, 0.996666669845581,
        0.995555579662323, 0.9800000190734863, 0.9977777600288391, 1.0,
        0.9988889098167419, 0.9933333396911621]
```

```
time_3 = [89.465306282, 89.588342667, 90.659433365, 91.204991579,
    91.249099970, 90.745548010, 88.521045923, 88.634417295,
    88.166294336, 88.489586830]
time_9 = [188.675819397, 187.791188002, 188.838074684, 188.192988634,
    187.358687162, 189.160743952, 187.565135241, 187.414045334,
    186.051147461, 184.723701000]

accuracy_dark = [1.0, 0.9800000190734863, 0.9888888597488403,
    0.9900000095367432, 0.9855555295944214, 1.0, 0.9988889098167419,
    0.9888888597488403, 0.9988889098167419, 1.0]
accuracy_light = [0.6600000262260437, 0.6888889074325562,
    0.5266666412353516, 0.8322222232818604, 0.6411111354827881,
    0.7133333086967468, 0.5666666626930237, 0.7900000214576721,
    0.6688888669013977, 0.5022222399711609]
time_dark = [91.165779829, 91.832492590, 90.854109049, 92.614001751,
    92.014000177, 91.847000360, 92.895925760, 92.011271000,
    94.807870626, 92.280503273]
time_light = [61.931142330, 62.510395050, 63.989901304, 63.338240385,
    63.023599625, 62.553331137, 63.672000647, 62.559733391,
    61.743544102, 61.475999594]

compare_distributions(accuracy_32, accuracy_64, time_32, time_64)
compare_distributions(accuracy_3, accuracy_9, time_3, time_9)
compare_distributions(accuracy_dark, accuracy_light, time_dark,
    time_light)
```

Bibliography

1. P. F. McManamon, *LiDAR Technologies and Systems*. Bellingham, Washington, USA: SPIE Press, July 2019.

2. M.-C. Amann, T. M. Bosch, M. Lescure, R. A. Myllylae, and M. Rioux, "Laser ranging: a critical review of unusual techniques for distance measurement," *Optical Engineering*, vol. 40, pp. 10–19, Jan. 2001.

3. K. Marcoe, "LIDAR an Introduction and Overview," 2007. Accessed: 03 Jan 22 [Online]. Available: http://web.pdx.edu/~jduh/courses/Archive/geog481w07/Students/Marcoe_LiDAR.pdf.

4. P. F. McManamon, G. Kamerman, and M. Huffaker, "A history of laser radar in the United States," in *Laser Radar Technology and Applications XV* (M. D. Turner and G. W. Kamerman, eds.), vol. 7684, pp. 256 – 266, International Society for Optics and Photonics, SPIE, Apr. 2010.

5. V. Molebny, G. Kamerman, and O. Steinvall, "Laser radar: from early history to new trends," in *Electro-Optical Remote Sensing, Photonic Technologies, and Applications IV* (G. W. Kamerman, O. Steinvall, K. L. Lewis, R. C. Hollins, T. J. Merlet, G. J. Bishop, and J. D. Gonglewski, eds.), vol. 7835, pp. 9 – 38, International Society for Optics and Photonics, SPIE, Oct. 2010.

6. L. Vonasek, "3D Live Scanner," Oct. 2021. Accessed: 03 Jan 22 [Online]. Available: https://play.google.com/store/apps/details?id=com.lvonasek.arcore3dscanner.

7. S. Rangwala, "The iPhone 12 - LiDAR At Your Fingertips," Nov. 2020. Accessed: 08 Jun 21 [Online]. Available: https://www.forbes.com/sites/sabbirrangwala/2020/11/12/the-iphone-12lidar-at-your-fingertips/?sh=2bba42e03e28.

8. L. Vonasek, "ToF Viewer / Night Vision," Mar. 2021. Accessed: 04 Jan 22 [Online]. Available: https://play.google.com/store/apps/details?id=com.lvonasek.tofviewer&hl=en_US&gl=US.

9. A. Chilton, "The Working Principle and Key Applications of Infrared Sensors," Oct. 2014. Accessed: 04 Jan 22 [Online]. Available: https://www.azosensors.com/article.aspx?ArticleID=339.

10. S. K. Naya, K. Ikeuch, and T. Kanade, "Surface Reflection: Physical and Geometrical Perspectives," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 13, pp. 611–634, July 1991.

11. J. A. Stuchi, L. Boccato, and R. Attux, "Frequency learning for image classification," June 2020. Accessed: 13 Jun 21 [Online]. Available: https://arxiv.org/abs/2006.15476v1.

12. T. Raj, F. H. Hashim, A. B. Huddin, M. F. Ibrahim, and A. Hussain, "A survey on lidar scanning mechanisms," *Electronics*, vol. 9, no. 5, p. 741, 2020.

13. M. Vollmer, K.-P. Möllmann, and J. A. Shaw, "The optics and physics of near infrared imaging," *Education and Training in Optics and Photonics: ETOP 2015*, vol. 9793, pp. 174 – 181, Oct. 2015.

14. S. W. Wang, J. C. Lai, Z. J. Li, X. D. Yang, and Z. H. Li, "Design and implementation of a linear array laser emitting optical system based on diffractive principles," *Optik*, vol. 224, p. 165693, Dec. 2020.

15. L. Li, "Time-of-Flight Camera-An Introduction," tech. rep., Texas Instruments, Jan. 2014. Accessed: 13 Jun 21 [Online]. Available: https://www.ti.com/lit/wp/sloa190b/sloa190b.pdf?ts=1641991552897.

16. M. Dummer, K. Johnson, S. Rothwell, K. Tatah, and M. Hibbs-Brenner, "The role of VCSELs in 3D sensing and LiDAR," in *Optical Interconnects XXI* (H. Schröder and R. T. Chen, eds.), vol. 11692, pp. 42 – 55, International Society for Optics and Photonics, SPIE, 2021.

17. C. Bruschini, H. Homulle, I. M. Antolovic, S. Burri, and E. Charbon, "Single-photon avalanche diode imagers in biophotonics: review and outlook," *Light: Science and Applications*, vol. 8, pp. 2047–7538, Dec. 2019.

18. E. R. Fossum, "CMOS Image Sensors: Electronic Camera-On-A-Chip," *IEEE Transactions on Electron Devices*, vol. 44, no. 10, pp. 1689–1698, 1997.

19. N. Akahane, S. Sugawa, S. Adachi, and K. Mizobuchi, "Wide dynamic range CMOS image sensors for high quality digital camera, security, automotive and medical applications," in *Proceedings of IEEE Sensors*, pp. 396–399, 2006.

20. V. Akopyan, "A modified DSLR does not make a good multispectral sensor," July 2016. Accessed: 08 Jun 21 [Online]. Available: https://blog.quickbird.uk/why-a-modified-dslr-does-not-make-a-good-multispectral-sensor-6766a9270820.

21. K. Bowyer, "Day/night IP cameras: What is an infrared-cut filter?," Sept. 2012. Accessed: 22 Dec 21 [Online]. Available: https://www.networkwebcams.co.uk/blog/2012/09/18/day-night-switching-ir-cut-filters/.

22. S. C. Lauxtermann, A. O. Lee, J. P. Stevens, and A. Joshi, "Comparison of global shutter pixels for cmos image sensors," 2007. Accessed: 14 June 21 [Online]. Available: http://www.imagesensors.org/Past%20Workshops/2007%20Workshop/2007%20Papers/021%20Lauxtermann%20et%20al.pdf.

23. "4K Ultra HD Resolution 8MP Outdoor IP Camera, 200ft Night Vision." Accessed: 14 June 21 [Online]. Available: https://www.lorextechnology.com/4k-security-camera/4k-ultra-hd-8mp-noctunnal-ip-camera/LNB8921BW-1-p.

24. M. Y, "Apple LIDAR Demystified: SPAD, VCSEL, and Fusion. . . ," Mar. 2021. Accessed: 08 Jun 21 [Online]. Available: https://4sense.medium.com/apple-lidar-demystified-spad-vcsel-and-fusion-aa9c3519d4cb.

25. I. Team, "Frame Rate Guide for Video Surveillance," Jan. 2021. Accessed: 14 Jun 21 [Online]. Available: https://ipvm.com/reports/frame-rate-surveillance-guide.

26. J. Chouinard, "The Fundamentals of Camera and Image Sensor Technology," 2019. Accessed: 22 Dec 21 [Online]. Available: https://www.visiononline.org/userassets/aiauploads/file/cvp_the-fundamentals-of-camera-and-image-sensor-technology_jon-chouinard.pdf.

27. J. F. Kurose and K. W. Ross, *Computer Networking : A Top-Down Approach*. Pearson, 8 ed., 2020.

28. "Simple guide of IP camera bitrate setting," July 2015. Accessed: 29 Dec 21 [Online]. Available: https://www.unifore.net/ip-video-surveillance/simple-guide-of-ip-camera-bitrate-setting.html.

29. D. Djudjic, "This video helps you understand the rolling shutter effect," July 2017. Accessed: 23 Dec 21 [Online]. Available: https://www.diyphotography.net/this-video-helps-you-understand-the-rolling-shutter-effect/.

30. Lorex, "4K Ultra HD IR Bullet IP Camera LNB8921_QSG_EN_R2 Quick Start Guide English Version 2.0," 2018. Accessed: 31 Dec 21 [Online]. Available: https://www.lorextechnology.com/downloads/ip-cameras/LNB8921/LNB8921_QSG_EN_R2_web.pdf.

31. Any-Lamp, "What are Lumen and Lux? ." Accessed: 04 Jan 22 [Online]. Available: https://www.any-lamp.com/blog/lumen-and-lux.

32. R. C. Gonzalez and R. E. Woods, *Digital Image Processing*. New York City: Pearson, 4th ed., Mar. 2017.

33. B. Chitradevi and P. Srimathi, "An Overview on Image Processing Techniques," *International Journal of Innovative Research in Computer and Communication Engineering*, vol. 2, pp. 6466–6472, Nov. 2014.

34. N. Otsu, "A Threshold Selection Method from Gray-Level Histograms," *IEEE Trans Syst Man Cybern*, vol. SMC-9, no. 1, pp. 62–66, 1979.

35. T. Trnovszký, P. Sýkora, and R. Hudec, "Comparison of Background Subtraction Methods on Near Infra-Red Spectrum Video Sequences," *Procedia Engineering*, vol. 192, pp. 887–892, Jan. 2017.

36. I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016.

37. D. P. Kingma and J. L. Ba, "Adam: A Method for Stochastic Optimization," Dec. 2017. Accessed: 30 Dec 21 [Online]. Available: https://arxiv.org/abs/1412.6980v9.

38. P. Zhou, J. Feng, C. Ma, C. Xiong, S. Hoi, and W. E, "Towards Theoretically Understanding Why SGD Generalizes Better Than ADAM in Deep Learning," Nov. 2021. Accessed: 30 Dec 21 [Online]. Available: https://arxiv.org/abs/2010.05627.

39. "Python." Accessed: 31 Dec 21 [Online]. Available: https://www.python.org/.

40. "PyCharm: the Python IDE for Professional Developers by JetBrains." Accessed: 31 Dec 21 [Online]. Available: https://www.jetbrains.com/pycharm/.

41. "Conda." Accessed: 31 Dec 21 [Online]. Available: https://docs.conda.io/en/latest/.

42. "pip." Accessed: 31 Dec 21 [Online]. Available: https://pip.pypa.io/en/stable/.

43. "OpenCV." Accessed: 31 Dec 21 [Online]. Available: https://opencv.org/.

44. "TensorFlow." Accessed: 31 Dec 21 [Online]. Available: https://www.tensorflow.org/.

45. "NumPy." Accessed: 31 Dec 21 [Online]. Available: https://numpy.org/.

46. S. van der Walt, J. L. Schönberger, J. Nunez-Iglesias, F. Boulogne, J. D. Warner, N. Yager, E. Gouillart, T. Yu, and the scikit-image contributors, "scikit-image: image processing in Python," *PeerJ*, vol. 2, p. e453, June 2014.

47. PyImageSearch, "imutils," Jan. 2021. Accessed: 03 Jan 22 [Online]. Available: https://github.com/PyImageSearch/imutils.

48. Python Software Foundation, "Python 3.9.9 Documentation," 2022. Accessed: 03 Jan 22 [Online]. Available: https://docs.python.org/3.9/.

49. N. W. Team, "NACL Web Plug-in," 2015. Accessed: 14 Jun 21 [Online]. Available: https://chrome.google.com/webstore/detail/nacl-web-plug-in/pbdcmagkbhnjpjlnpibbmggikpedpilc.

50. "Swift." Accessed: 31 Dec 21 [Online]. Available: https://developer.apple.com/swift/.

51. "Xcode 13 Overview." Accessed: 31 Dec 21 [Online]. Available: https://developer.apple.com/xcode/.

52. Apple, "iPhone 12 Pro Max, iPhone 12 mini, and HomePod mini available to order Friday," Nov. 2020. Accessed: 03 Jan 21 [Online]. Available: https://www.apple.com/newsroom/2020/11/iphone-12-pro-max-iphone-12-mini-and-homepod-mini-available-to-order-friday/.

53. Samsung, "Galaxy S20, S20+ and S20 Ultra 5G," 2022. Accessed: 03 Jan 21 [Online]. Available: https://www.samsung.com/us/mobile/galaxy-s20-5g/specs/.

54. HUAWEI, "HUAWEI P40 Pro," 2022. Accessed: 03 Jan 21 [Online]. Available: https://consumer.huawei.com/en/phones/p40-pro/.

55. Samsung, "What is ToF camera technology on Galaxy and how does it work?." Accessed: 03 Jan 21 [Online]. Available: https://www.samsung.com/global/galaxy/what-is/tof-camera/.

56. Flowsome, "P40 Series: TOF Camera: Inconspicuous but Indispensable ∼ Give it Space to Breathe!," June 2020. Accessed: 03 Jan 21 [Online]. Available: https://consumer.huawei.com/ph/community/details/P40-Series-TOF-Camera-Inconspicuous-but-Indispensable-Give-it-Space-to-Breathe/topicId_106309/.

57. Google, "Google Play," 2022. Accessed: 03 Jan 21 [Online]. Available: https://play.google.com/store?hl=en_US&gl=US.

58. Apple, "App Store," 2022. Accessed: 03 Jan 21 [Online]. Available: https://www.apple.com/app-store/.

59. HUAWEI, "HUAWEI AppGallery," 2022. Accessed: 03 Jan 21 [Online]. Available: https://appgallery.huawei.com/Featured.

60. L. Vonasek, "3D Live Scanner," 2021. Accessed: 03 Jan 22 [Online]. Available: https://appgallery.huawei.com/#/app/C103316231.

61. Apple, "Apple Developer," 2022. Accessed: 03 Jan 21 [Online]. Available: https://developer.apple.com/.

62. "Lux Meter Apps: Why Most of Them are Useless." Accessed: 31 Dec 21 [Online]. Available: https://growlightmeter.com/lux-meter-apps-why-id-only-use-photone/.

63. V. Molebny, P. Zarubin, and G. Kamerman, "The dawn of optical radar: a story from another side of the globe," in *Laser Radar Technology and Applications XV* (M. D. Turner and G. W. Kamerman, eds.), vol. 7684, pp. 96 – 105, International Society for Optics and Photonics, SPIE, Apr. 2010.

64. J. U. Eitel, B. Höfle, L. A. Vierling, A. Abellán, G. P. Asner, J. S. Deems, C. L. Glennie, P. C. Joerg, A. L. LeWinter, T. S. Magney, G. Mandlburger, D. C. Morton, J. Müller, and K. T. Vierling, "Beyond 3-D: The new spectrum of lidar applications for earth and ecological sciences," *Remote Sensing of Environment*, vol. 186, pp. 372–392, Dec. 2016.

65. S. Bailey, W. McKeag, J. Wang, M. Jack, and F. Amzajerdian, "Advances in HgCdTe APDs and LADAR receivers," in *Infrared Technology and Applications XXXVI* (B. F. Andresen, G. F. Fulop, and P. R. Norton, eds.), vol. 7660, pp. 1101 – 1113, International Society for Optics and Photonics, SPIE, Apr. 2010.

66. E. de Borniol, F. Guellec, J. Rothman, A. Perez, J.-P. Zanatta, M. Tchagaspanian, P. Castelein, G. Destéfanis, J.-C. Peyrard, and F. Pistone, "HgCdTe-based APD focal plane array for 2D and 3D active imaging: first results on a 320 x 256 with 30 µm pitch demonstrator," in *Infrared Technology and Applications XXXVI* (B. F. Andresen, G. F. Fulop, and P. R. Norton, eds.), vol. 7660, pp. 1067 – 1075, International Society for Optics and Photonics, SPIE, 2010.

67. A. Ashcroft and I. Baker, "Developments in HgCdTe avalanche photodiode technology and applications," in *Infrared Technology and Applications XXXVI* (B. F. Andresen, G. F. Fulop, and P. R. Norton, eds.), vol. 7660, pp. 1058 – 1066, International Society for Optics and Photonics, SPIE, 2010.

68. T. Tether, "Subcommittee on Terrorism, Unconventional Threats and Capabilities, House Armed Services Committee, U.S. House of Representatives," Mar. 2004. Accessed: 14 Jun 21 [Online]. Available: https://www.darpa.mil/attachments/TestimonyArchived(March252004).pdf.

69. "HDL-64E Durable Surround Lidar Sensor — Velodyne Lidar." Accessed: 14 Jun 21 [Online]. Available: https://velodynelidar.com/products/hdl-64e/.

70. M. Vollmer and K. P. Möllmann, *Infrared Thermal Imaging: Fundamentals, Research and Applications*. Weinheim, Germany: Wiley-VCH, 2nd ed., Nov. 2017.

71. C. L. Herry, M. Frize, and R. A. Goubran, "Segmentation and Landmark Identification in Infrared Images of the Human Body," in *International Conference of the IEEE Engineering in Medicine and Biology*, (New York, NY, USA), pp. 957–960, Institute of Electrical and Electronics Engineers, 2006.

72. P. Zhang, H. Ji, H. Wang, Y. Liu, X. Zhang, and C. Ren, "Quantitative evaluation of impact damage to apples using NIR hyperspectral imaging," *International Journal of Food Properties*, vol. 24, no. 1, pp. 457–470, 2021.

73. J. Y. Barnaby, T. D. Huggins, H. Lee, A. M. McClung, S. R. Pinson, M. Oh, G. R. Bauchan, L. Tarpley, K. Lee, M. S. Kim, and J. D. Edwards, "Vis/NIR

hyperspectral imaging distinguishes sub-population, production environment, and physicochemical grain properties in rice," *Scientific Reports*, vol. 10, pp. 1–13, June 2020.

74. C. Liu, W. Liu, X. Lu, W. Chen, J. Yang, and L. Zheng, "Nondestructive determination of transgenic Bacillus thuringiensis rice seeds (Oryza sativa L.) using multispectral imaging and chemometric methods," *Food Chemistry*, vol. 153, pp. 87–93, 2014.

75. X. Dai, Y. Duan, J. Hu, S. Liu, C. Hu, Y. He, D. Chen, C. Luo, and J. Meng, "Near infrared nighttime road pedestrians recognition based on convolutional neural network," *Infrared Physics & Technology*, vol. 97, pp. 25–32, Mar. 2019.

76. S. Chung, L. E. Breshears, and J.-Y. Yoon, "Smartphone near infrared monitoring of plant stress," *Computers & Electronics in Agriculture*, vol. 154, pp. 93–98, Sept. 2018.

77. T. Sakamoto, A. A. Gitelson, A. L. Nguy-Robertson, T. J. Arkebauer, B. D. Wardlow, A. E. Suyker, S. B. Verma, and M. Shibayama, "An alternative method using digital cameras for continuous monitoring of crop status," *Agricultural and Forest Meteorology*, vol. 154-155, pp. 113–126, Mar. 2012.

78. G. D. Smith, E. Nunan, C. Walker, and D. Kushel, "Inexpensive, near-infrared imaging of artwork using a night-vision webcam for chemistry-of-art courses," *Journal of Chemical Education*, vol. 86, pp. 1382–1388, Dec. 2009.

79. H. V. Dantas, M. F. Barbosa, A. Pereira, M. J. C. Pontes, P. N. T. Moreira, and M. C. U. Araújo, "An inexpensive NIR LED Webcam photometer for detection of adulterations in hydrated ethyl alcohol fuel," *Microchemical Journal*, vol. 135, pp. 148–152, Sept. 2017.

80. A. R. Petach, M. Toomey, D. M. Aubrecht, and A. D. Richardson, "Monitoring vegetation phenology using an infrared-enabled security camera," *Agricultural and Forest Meteorology*, vol. 195-196, pp. 143–151, Sept. 2014.

81. M. Ju, J. Luo, P. Zhang, M. He, and H. Luo, "A Simple and Efficient Network for Small Target Detection," *IEEE Access*, vol. 7, pp. 85771–85781, June 2019.

82. P. Li, B. Yan, R. Ye, and G. Sun, "An infrared dim and small target detection method based on fractional differential," *Proceedings of the 30th Chinese Control and Decision Conference*, pp. 2381–2386, July 2018.

83. Y. Chen and Y. Xin, "An Efficient Infrared Small Target Detection Method Based on Visual Contrast Mechanism," *IEEE Geoscience and Remote Sensing Letters*, vol. 13, pp. 962–966, July 2016.

84. J. Gauci, O. Falzon, K. P. Camilleri, C. Formosa, A. Gatt, C. Ellul, S. Mizzi, A. Mizzi, K. Cassar, C. Sturgeon, and N. Chockalingam, "Automated segmentation of regions of interest from thermal images of hands," in *39th Annual International Conference of the IEEE Engineering in Medicine and Biology Society*, (Jeju, South Korea), pp. 3822–3826, Institute of Electrical and Electronics Engineers, Sept. 2017.

85. T. H. Yoon, K. S. Kim, J. W. Lee, D. J. Kim, and C. G. Song, "Image segmentation of human forearms in infrared image," in *Annual International Conference of the IEEE Engineering in Medicine and Biology*, (New York, NY, USA), pp. 2762–2765, Institute of Electrical and Electronics Engineers, 2006.

86. M. R. Haque and S. Z. Mishu, "Spectral-spatial feature extraction using pca and multi-scale deep convolutional neural network for hyperspectral image classification," in *22nd International Conference on Computer and Information Technology*, pp. 1–6, 2019.

87. Y. Liu, Z. Zhang, X. Liu, L. Wang, and X. Xia, "Deep learning-based image classification for online multi-coal and multi-class sorting," *Computers & Geosciences*, vol. 157, p. 104922, 2021.

88. J. Cheng, S. Tian, L. Yu, C. Gao, X. Kang, X. Ma, W. Wu, S. Liu, and H. Lu, "ResGANet: Residual group attention network for medical image classification and segmentation," *Medical Image Analysis*, vol. 76, p. 102313, 2022.

89. R. L. Rocha, C. D. Silva, A. C. S. Gomes, B. V. Ferreira, E. C. Carvalho, A. C. Q. Siravenha, and C. C. Rosa, "CNN-DFT Based Approach Applied to Image Inspection of Railcar Component: A Comparison with Machine Learning Methods," *Journal of Information and Data Management*, vol. 11, pp. 20–33, June 2020.

90. A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," *Communications of the ACM*, vol. 60, pp. 84–90, June 2017.

91. F. Chollet, "Xception: Deep Learning with Depthwise Separable Convolutions," Apr. 2017. Accessed: 30 Dec 21 [Online]. Available: https://arxiv.org/abs/1610.02357.

92. C. M. Caridade, A. R. Marçal, and T. Mendonça, "The use of texture for image classification of black and white air photographs," *International Journal of Remote Sensing*, vol. 29, pp. 593–607, Jan. 2008.

93. TRENDnet, "5-Port 10/100 Mbps GREENnet Switch." Accessed: 31 Dec 21 [Online]. Available: https://www.trendnet.com/products/product-detail?prod=580_TE100-S5.

94. B. Security, "POE-SW502." Accessed: 31 Dec 21 [Online]. Available: https://bvsecurity.com/product/poe-sw502/.

95. ConceptDraw, "Cisco Network Topology Workstation Vector Stencil." Accessed: 31 Dec 21 [Online]. Available: https://conceptdraw.com/a1774c3/p45/preview/640/pict--workstation-cisco-network-topology-vector-stencils-library.

96. H. Hill, "Hard Disk Drive Picture Free Transparent Image HD," Apr. 2018. Accessed: 03 Jan 22 [Online]. Available: https://freepngimg.com/png/51022-hard-disk-drive-picture-free-transparent-image-hd.

97. Nathancy, "Storing RTSP stream as video file with OpenCV VideoWriter," 2019. Accessed: 10 Oct 21 [Online]. Available: https://stackoverflow.com/questions/55141315/storing-rtsp-stream-as-video-file-with-opencv-videowriter.

98. "Displaying a Point Cloud Using Scene Depth," 2020. Accessed: 10 Nov 21 [Online]. Available: https://developer.apple.com/documentation/arkit/environmental_analysis/displaying_a_point_cloud_using_scene_depth.

99. F. Sani and J. Todman, *Experimental Design and Statistics for Psychology: A First Course*, ch. Appendix 1, pp. 183–196. Malden, MA, USA: Blackwell Publishing Ltd., Jan. 2006.

100. F. Wilcoxon, "Individual Comparisons by Ranking Methods," *Biometrics Bulletin*, vol. 1, p. 80, Dec. 1945.

101. G. Divine, H. J. Norton, R. Hunt, and J. Dienemann, "A Review of Analysis and Sample Size Calculation Considerations for Wilcoxon Tests," *Anesthesia and Analgesia*, vol. 117, pp. 699–710, Sept. 2013.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704–0188

1. REPORT DATE *(DD–MM–YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From — To)*
24–03–2022	Master's Thesis	Sept 2020 — Mar 2022

4. TITLE AND SUBTITLE

Detection and Identification of Cellphone Emitted Light Detection and Ranging Light in Security Camera Video Footage

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Tristan V. Creek

5d. PROJECT NUMBER

21G532A

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Air Force Institute of Technology
Graduate School of Engineering and Management (AFIT/EN)
2950 Hobson Way
WPAFB OH 45433-7765

8. PERFORMING ORGANIZATION REPORT NUMBER

AFIT-ENG-MS-22-M-019

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

AFRL/RYAA
2241 Avionic Cir
WPAFB OH 45433-7765
COMM 937-713-8573
Email: Eric.Lam.3@us.af.mil

10. SPONSOR/MONITOR'S ACRONYM(S)

AFRL/RYAA

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT

DISTRIBUTION STATEMENT A:
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

13. SUPPLEMENTARY NOTES

14. ABSTRACT The prevalence of Light Detection and Ranging (LiDAR) sensors in consumer cellphones has ushered in an era of access to technologies previously inaccessible to the average person. Although LiDAR sensors emit light in the infrared spectrum invisible to the naked eye, many security cameras possess the ability to capture infrared light reflected off surfaces to detect LiDAR light in an environment. Once detected and recorded to video footage, current classification methods fail to identify the LiDAR light. Therefore, this research develops a methodology to detect LiDAR light in security camera video footage and identify it as cellphone LiDAR. A proposed image processing technique separates the LiDAR light pattern from the background of a video frame. The extracted LiDAR light pattern is then permuted many ways to simulate how it may appear in video footage. A neural network trains to identify these permutations which results in a neural network that classifies whether or not a given image contains the LiDAR light pattern. In a dark environment with a cellphone and security camera positioned 1 meter from a wall, the trained neural network correctly identifies 99.7% of images containing cellphone LiDAR reflected off the wall.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Dr. Barry E. Mullins, AFIT/ENG
U	U	U	UU	210	19b. TELEPHONE NUMBER *(include area code)* (937) 255-3636, ext 7979; barry.mullins@afit.edu