MITS Professional Project

BIKE TRAFFIC DETECTION USING A CUSTOM, LOW POWER, LOW COST, MACHINE LEARNING BASED SENSOR

Prepared By
Riley Engle
Student
Ohio University
J. Warren McClure School of Information and Telecommunication Systems

Supervised By
Dr. Julio Arauz PhD
Graduate Director
Ohio University
J. Warren McClure School of Information and Telecommunication Systems

January 2018 – April 2018

Table of Contents

List of Tables

List of Figures

1. Introduction

This project aims to create a low-cost, low-powered sensor prototype capable of identifying bicycles. It accomplishes this by using a Particle Photon microcontroller with a serial camera with SD card to save and send images. Once an image is taken, a trained TensorFlow instance is used to classify the image, and identify what the image contains. All code produced in this project is available for replication, and locations of instructions to construct the sensor are also contained within this document. A focus was placed on being low-cost, low-power, and Internet-enabled with the purpose of making data collection feasible for small cities and towns to determine where bike-related infrastructure should be built. The goal of this project is to get a prototype of the sensor running that is low cost, low power, and utilizes a machine learning algorithm to classify images.

1.1 Rationale

IoT devices are quickly becoming a larger part of our lives. More and more consumer electronics are connected, allowing for information to be gathered about the device or its environment easily. IoT plays a large role in the design and creation of smart cities, where smart IoT sensors are placed within the city to better use resources or measure characteristics about how the city functions. Devices can be embedded into roads to measure traffic flows, put into buildings to measure water or electricity usage for devices, or measure air quality to relay information to citizens. The devices can also give valuable data to city administrators who want to invest infrastructure capital to better provide for citizens. Without certain data, cities unknowingly may be not providing resources that citizens need. Traditional methods of gathering data can be too expensive for some cities, which could be why they may be unable to implement the changes that benefit the citizens and city.

Athens County, Ohio is one of the most poverty-stricken regions in the state. According to DataUSA, it has a poverty rate of 54.7% [1]. Work done by the Buckeye Hills Regional Council suggests that 8.4% of citizens are unemployed, and nearly 45,000 people leave the southeastern Ohio region to find work, with a jobs deficit of about 25,000 jobs [2]. The region has 6% of households identified as zero-car households, with the highest concentration within Athens county. This may force residents to travel via another method to travel, namely travelling by bicycle. However, roads that are travelled in the region may not always have bicyclists in mind when they were constructed. Cities in the region may want to invest infrastructure to give bicyclers a safer method of transportation. As mentioned above, this can be expensive, and even more so for the poverty-stricken regions of southeast Ohio.

1.2 Elements Out-Of-Scope

Constructing a new sensor to tackle this problem is difficult and involves numerous challenges, and as a result, several items are deemed to be out-of-scope for this project. This first iteration is a prototype that proves the concept is possible. Further refinements will be needed before deploying the sensor and improvements can be made.

Actual deployment of sensors is never addressed, and can be left to future researchers to find the best way or locations to deploy a refined version of the sensors. Locations were not examined because of time limitations, as well as to allow for flexibility to create a sensor without conforming to a specific design to fit a specific location.

Power optimization is also not examined in this project, since the devices that were chosen use power that is low enough to be powered by a battery for an extended period of time. The size of the battery will determine the operation time, but power usage can be improved to get the device to function longer.

Code optimization is not examined in this project. A functioning prototype was the intended end goal of this project, and as a result, getting code to work is the first goal. Improving code to run faster or more efficiently can be done by future researchers. Also, code can be customized, allowing for more features or functions to be added to the device, which could improve overall sensor performance.

Detection optimization is not examined in this paper. A functioning prototype may do well at detecting an object, but improvements could always be made to make the sensor detect every object. Detection for this project is aimed at getting a functioning device, not for perfecting every interaction to detect an object accurately.

Alternate uses for the sensor are also not examined in this project. The focus of this project is to detect bicycles, but the underlying technology that makes it detect bicycles – the TensorFlow machine learning program – is customizable and retrainable. The hardware of the device can be used for a variety of purposes if the corresponding TensorFlow instance is trained accordingly.

The TensorFlow algorithm can still be improved. The project is an ongoing, open-source project that is currently active. Improvements can be made, and most likely will be as time goes on. The instance of TensorFlow that was used in this project has been deemed the most up-to-date at the time the research was completed. Future versions may have improvements for this purpose that do not currently exist.

Mesh connectivity is not examined in this project. The technology to do a mesh connected network is certainly possible, but is not used in this project. A mesh network would limit the number of devices that need an actual Wi-Fi connection to send data. A mesh network would also send data to the next node,

until a gateway is reached, and the data is sent at that point. This could lead to a larger deployment area, but future researcher may wish to explore the use of mesh networks in this case.

The testing site that was used to conduct the experiments may be biased in some way. The location was a parking lot, where cars may have moved in and out. This caused the layout of the site to change, which could have led to incorrect detections or classifications. Also, the testing site could have had some characteristics that do not fit with data used to train the machine learning algorithms. This again could lead to incorrect classifications. Also, extensive testing of the sensor is not conducted. A simple experiment is run to examine how the prototype functions. More extensive testing is considered to be out of scope.

Alternate platforms were not examined in this project once a particular platform was chosen. However, there is no reason a researcher could not use an Arduino, ESP8266, ESP32, Raspberry Pi, or other small microcontroller to complete this project. The same can be said about machine learning services as well. Amazon Rekognition is also available as a subscription service. Alternate platforms could perform this task better or worse than what was chosen in this project, and is left to a future researcher to determine which hardware and software components are best for the project.

Privacy is not within the scope of this project. All uses in this project are intended to be used in public situations. As a result, the images do not need consent to be taken, and are also not transported with privacy in mind. Securing the connection between the senor and the remote server was difficult to do, and the tasks on the remote server not related to the main goal could interfere with the function of the device due to memory limitations. This does not mean that privacy should not be a concern when using

this device, simply that the device did not have privacy as an item within the scope of the project, and could be implemented in future iterations of the sensor.

1.3 Document Structure

This document contains the information gained while completing the project. Motivations for completing the project are contained within the first section. A selection of offerings of bike sensors currently available on the market were examined as well. The current offerings were examined to get an idea of what technology is being used currently, as well as how these devices are being used to count bicycles. All products examined are available for purchase with accompanying software. System requirements that were identified over the course of constructing the sensor are detailed and the requirements could be used to select a new device for construction, or lead readers to understand why a particular path was chosen during the course of building the sensor. This document also contains locations of instructions on how to create this device, with wiring diagrams and a parts list available as well. Instructions on how to use the TensorFlow instance are also included and available. A detailed analysis of the code used is incorporated into this document as well, and will include a complete copy of code used to get the project functioning.

This document also contains the existing art that has been completed in this field. It examines an offering of common sensors that are purpose built for this purpose, and examines why they may or may not be a good choice for this problem. Next, a section is dedicated to outlining the motivations for building this sensor. Throughout this document there is information about the construction of this sensor so other future researchers may build their own sensors in a similar manner to this project. An outline of requirements was generated and then a prototype was built. An experiment was conducted to test how the sensor functioned, and its results are detailed and discussed.

2. Motivation

This section examines the Regional Transportation Planning Organization, a sub part of the Buckeye Hills Regional Council, and the long-range plan that was published in 2015 to examine current and future potential issues that may arise related to general transportation issues. It contains demographic data about the region, and information regarding traffic changes for the future.

The Buckeye Hills Regional Council represents 8 counties in Southeastern Ohio. The Regional Transportation Planning Organization, a sub-committee within the organization, conducted a research study of the region's traffic flows, road usage, and related demographics, past data, and other factors to develop a long-range plan for the region as related to transportation. Using this report, several motivations for this project could be identified.

The Buckeye Hills Regional Council created infographics with details about road functional class and where accidents occurred. Figure 1 is the diagram of functional class roads.

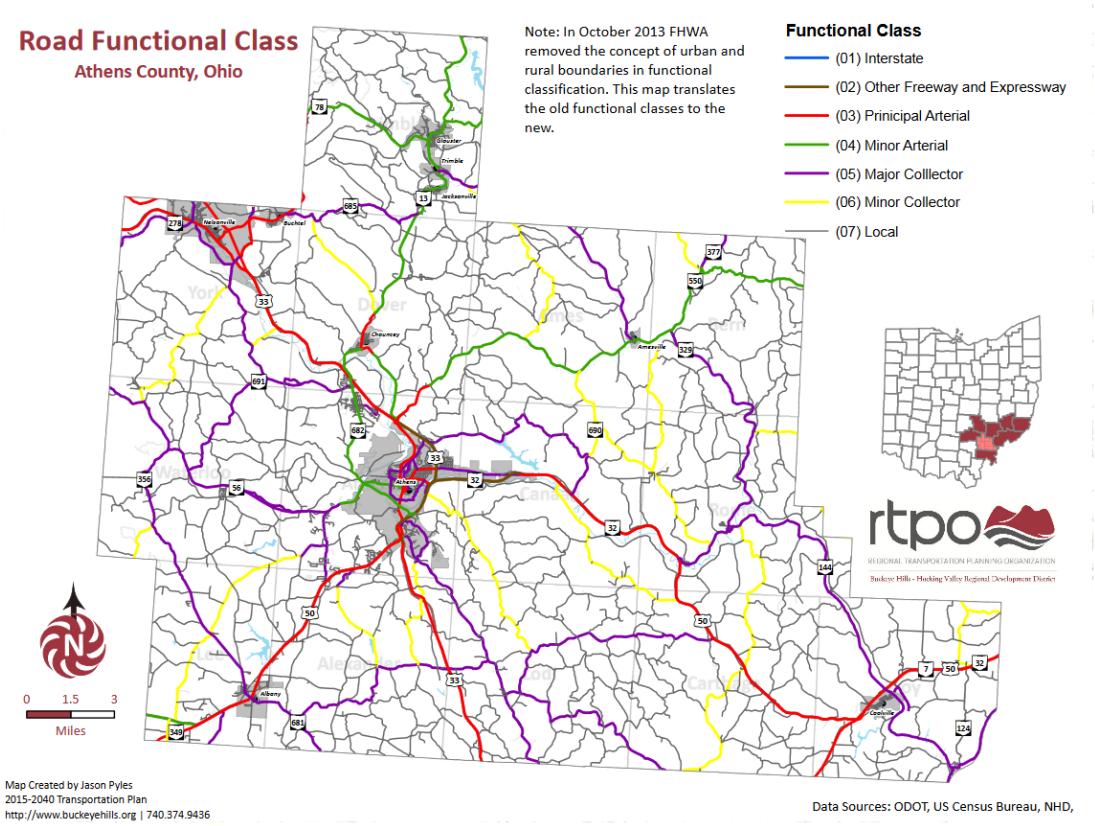


Figure 1 BHRC Long Range Plan: Athens County Road Functional Class

This figure outlines all major roads in the area. Roads outlined in grey are labeled as “local” roads. Major and minor collector roads are colored purple and yellow respectively. Local roads are often roads with the lowest speed limit, and carry much lower volumes of traffic than compared to other classes of roads. Collectors gather traffic from local roads and proved access to higher class roads, while also serving as a way for cars to go to and from other major roadways. Most cyclists could be found on these streets, potentially more on local roads than the other road classes. Even if a biker were to travel a large distance, they likely would not take major roads like Route 33 (outlined in red as a principal arterial class of road) for concerns of safety. Figure 2 details the accidents that occurred in Athens County between 2009 and 2013.

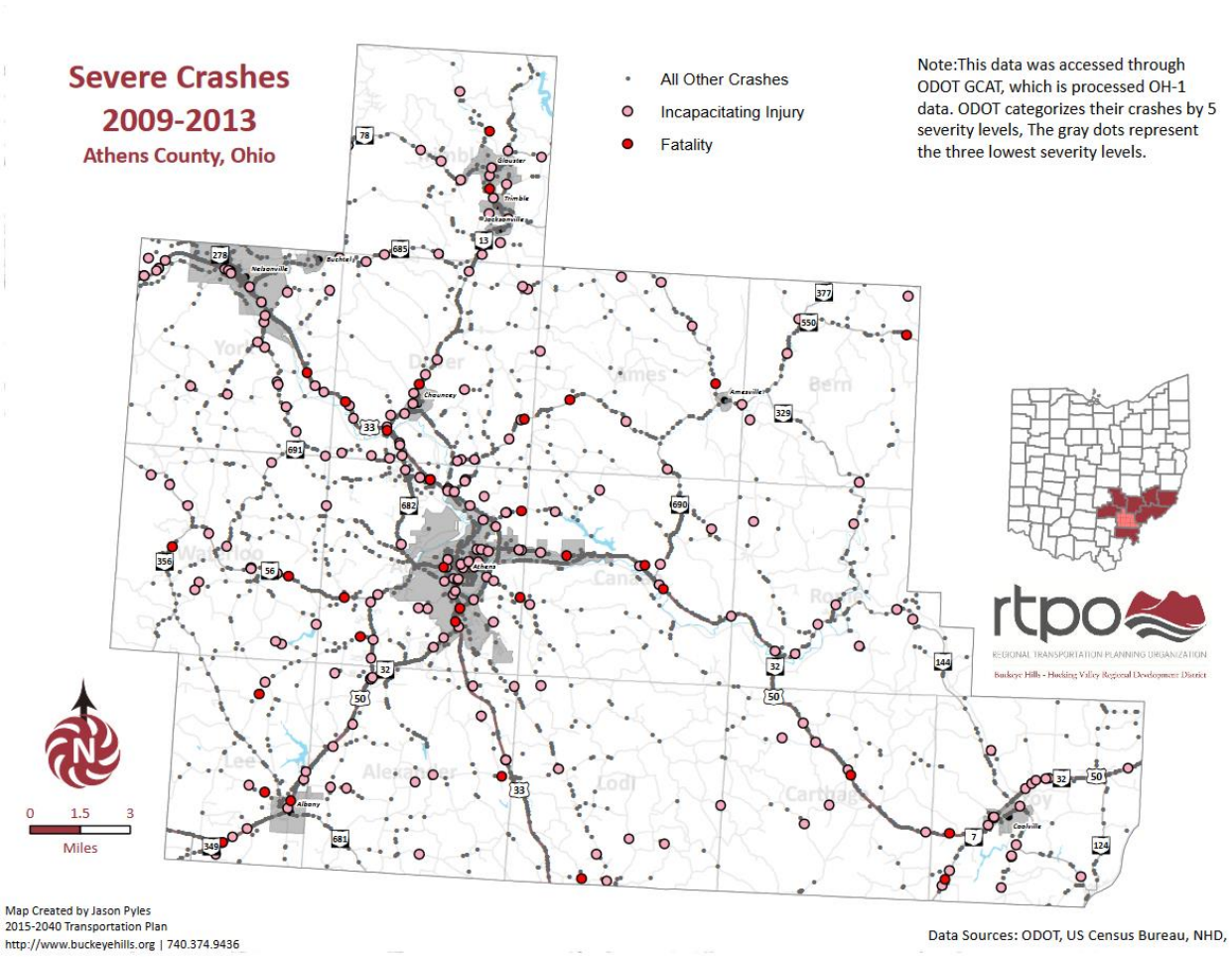


Figure 2 Crashes between 2009 and 2013 in the Buckeye Hills Region

This graphic show that many accidents occur on the higher classes of roads, yet a large number of accidents still occur on the local and collector roads. Statistically, some of these accidents are bicycle and pedestrian crashes.

Residents that are forced to use a bicycle and travel along roads with no sidewalks are 88% more likely at risk of an accident on a road with no separated sidewalks [3]. Almost 8% of all pedestrian deaths are due to along the roadway accidents [3]. 818 bicyclists were killed in accidents in 2015, and an estimated 45,000 biking-related injuries took place [4]. Of those that were killed, 71% took place in an urban setting, and

42% of injury causing accidents were directly related to vehicles or the roads the bicyclist was travelling on [3].

Safety is a motivator for this project, since safety can be increased by building new infrastructure for both pedestrians and bicycles. Knowing where to build infrastructure or how current infrastructure is being used is what a bike sensor aims to answer.

3. Existing Art – Current Sensor Offerings

Several bike counters currently exist on the market. They range in technology choices as well as price. Overall, options exist that range from manual counters to piezo electric strips, pneumatic tubes, and IR. Following are several examples of existing technologies

3.1 Eco Counter

The Eco Counter line of bicycle counting produces have several methods to count bicycles in a variety of regions. Their products include the Urban Zelt, Zelt Greenways, Easy Zelt, Eco-Combo, and Tube counter. Each method utilizes a different way to count bicycles. The Urban, Greenway, and Easy Zelt models both utilize an inductive loop that analyzes the electromagnetic signature of each bicycle's wheels as it travels over the sensor to accurately count the bike traffic [5]. The device is also hidden in the sidewalk, street, or path that it is deployed in for the Urban and Greenway Zelts, while the Easy Zelt sits on top of the surface where it is measuring traffic. These models are their flagship bicycle counters boasting a battery life of 2 years, industrial grade waterproofing, and automatic data transmission over the air with 3G, GSM, or Bluetooth communications available. The units mounted under the roadways are also nearly immune to vandalism, since they cannot be accessed from the outside [5]. The loops connect to a unit that records

the data. Figure 1 is an example drawing of the Zelt loops, while Figure 2 shows an installation in a roadway.

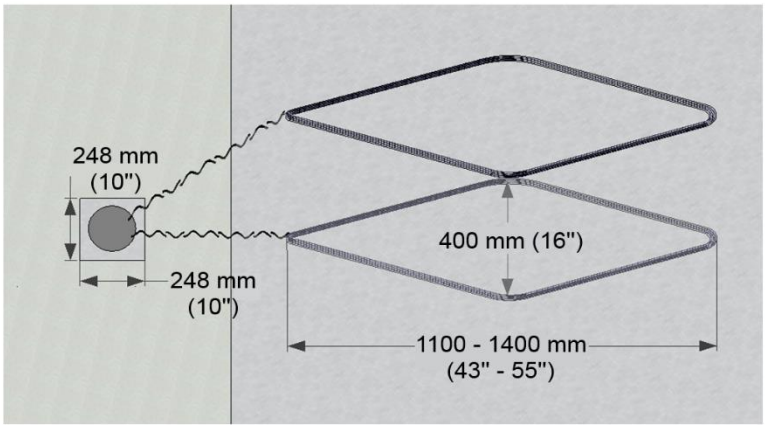


Figure 3 Zelt Inductive Loop Drawing

Figure 4 Eco-Counter Zelt Inductive Loop in Roadway

The tube counters offered by Eco-Counter are also available with waterproofing and a battery life of 10 years. It can store 2 years of data, recorded in 15-minute intervals. Data is saved to an accompanying unit that is attached to each end of the tubes [5]. Figure 3 shows the tubes on a street surface.

Figure 5 Eco-Counter Zelt Road Tubes on Roadway

The Eco-Combo MULTI is a device that can count both pedestrians and bicycles. This device is currently replaced by the unit called the Urban Post. It features a battery life ranging from 2 to 10 years, with either GSM or Bluetooth connections, waterproofing, and enough memory for about 1 to 2 years of data. This device mounts on a bollard or post to blend in with the surrounding environment [5]. Figure 4 is a 3d Render of the device showing the internal parts and external shell.

Figure 6 Eco-Counter Zelt Urban Post

These devices claim to have a high accuracy, as confirmed by Krista Nordback [6] in the experiments that were run with the new technology. According to Missoula County, as well as Nordback, these devices cost between $2,000 and $5,000 [6] [7]. The accompanying software, Eco-Visio is $420.00 as well per year, per device with automatic transmission [7].

3.2 Jamar Technologies, Inc.

Jamar Technologies is a company that specializes in collecting traffic data. They offer several solutions to bicycle counting specifically, as well as devices for other traffic uses, like vehicle distance data, report generation, and police radar recording.

Their line of bicycle sensors includes the TRAX Cycles Plus and TDC-Ultra. The TRAX Cycles Plus model devices come with road tubes that mount to the surface of the street, that then communicate with an accompanying unit to record data. It features GPS to record the location of the counts. It is designed for

mixed lane use, meaning it can differentiate bicycles from cars and other traffic. It can also utilize up to 4 road tubes as inputs. Figure 5 shows the unit that attaches to the road tubes to gather the data [8].

Figure 7 Jamar TRAX Cycles Plus

The other model that counts bicycles from Jamar is called the TDC Ultra. It is a device that allows one to electronically record data as bicycles pass by. It features the ability to record data to a user-determined granularity, up to 48 categories. This allows for details such as if the biker wears a helmet, or their gender. This device is handheld, and can capture a lot of detailed data electronically while a person is using it. It can then be uploaded to Jamar's PETRAPro software for data analysis. The device runs on 4 AA batteries. Figure 6 shows the TDC Ultra from the Jamar page for the product [8].

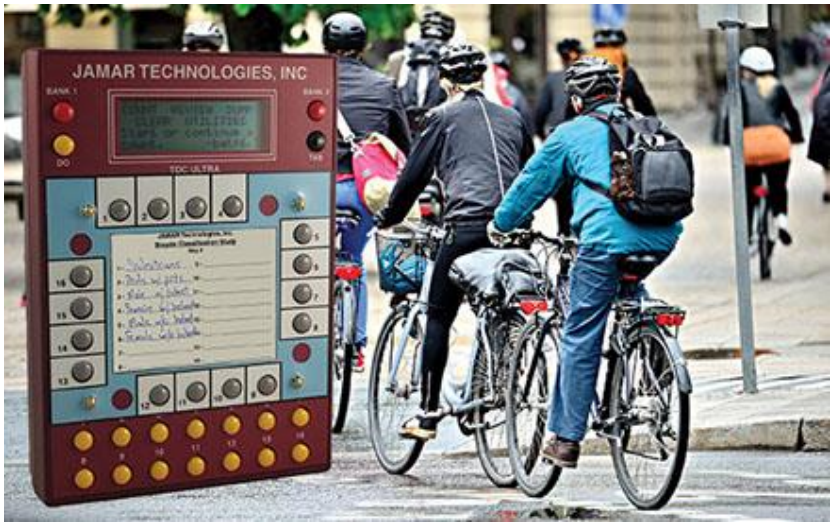


Figure 8 Jamar Technologies TDC ULTRA Electronic Counter

Jamar's devices range in cost. Pricing for the TRAX model device was not readily available from the device website but software for the TRAX line of devices cost $1495.00 for up to 5 users. The TDC ULTRA is available without software for $1095.00, and with a software license up to 5 seats for $1995.00. The TDC ULTRA must be used by a human operator, while the TRAX device can run automatically [8].

3.3 MetroCount

Metrocount is a company that offers several traffic data collection sensors. Their sensor for bicycles is called the RidePod BT. It utilizes road tubes and an accompanying road case and main unit to record the captured data. It is an improved version of the MC5620 Bicycle counter. It claims to be capable of recording bicycle volume, speed, direction, groups, and headway according to their website for the RidePod BT [9]The device is powered from 4 D batteries and can store data for up to 2 million bicycles. MetroCount also offers a device for measuring pedestrians and bicycles utilizing piezoelectric strips, called the RidePod BP, marketed as a more permanent solution. Both devices utilize a 6v, 18Ah battery that MetroCount also sells replacements for. A single battery is estimated to last between 220 days and 3 years. The sensor devices can mount to a steel cabinet enclosure near the sensor deployment location via

DIN rails, and can be locked in place for protection. The device can also be placed into a metal road case for protection as well.

MetroCount offers accompanying software that analyzes data via a purchasable service and software, MetroCount Traffic Executive. Data can be delivered via their FieldPod Service. and data is automatically sent via Email to specified recipients via 3G [10]. Figure 9 shows the RidePod BT and Figure 10 shows the cabinet enclosure, while Figure 11 shows the road case.

Figure 9 MetroCount RidePod BT

Figure 10 MetroCount Steel Cabinet Enclosure

Figure 11 MetroCount Road Case

MetroCount offers supplies to go with the units as well, and sells a Bike Monitoring Kit for $195.00 including road tubes and hardware to attach and mount the tubes. The MetroCount Traffic Executive software is available from their store for $450.00 per user. Pricing for units were not readily available on their website [10].

4. System Design

4.1 User Requirements

Solution should be low cost

Current offerings from the Existing Art section above range in cost for software and hardware from about $1000.00 to $5000.00. This may not be a lot of money for most cities. However, cities are also needing to allocate tens of thousands of dollars, up to millions of dollars, to pursue improving biking infrastructure [11]. Cities with more citizens, like San Francisco, can divert more money to these causes because of the high population and demand. Smaller cities cannot divert so many resources to an entire bike program, let alone just the cost of measuring bike traffic around the city. A prototype consisting of readily available

hardware should provide the basis of a solution that can be low cost while still accurately counting bicycles.

A city that has less financial resources may still desire or need to create new biking infrastructure, but the cost of doing so may off-put the start of the process. A low-cost solution can spur this investment for these cities.

Solution should have an accuracy similar to current offerings

According to Nordback, Many systems over and undercount bicycles for a variety of reasons. These reasons can include other travelers stepping or driving over the tubes, or by having bicycles riding close together respectively [12]. However, accuracies for inductive loop sensors, such as the Eco-Counter Zelt product, can range from 74% to 99% in normal use tests [6]. Special cases produce a wider range of errors. Despite this, there are many special cases where the sensors did as well as human counters. A solution should be able to identify nearly any bicycle configuration (tandem bikes, bikers side-by-side, one after another etc.) to render a totally accurate count in all special cases. The solution should be capable of identifying normal cases, as well as special cases.

Solution should be low power

Low-power battery operated devices make maintenance easy, since the device will not need to be opened for continuous operation. If the battery only needs replaced once every few years, then the cost to operate is lower as well. This can explain the high cost of a device, since it will continue to operate on a single charge for 2 or more years in some cases. A new solution should have power limitations and factors in mind, so that battery or solar power is an option if a solution should be deployed. A solution should be low-power, so batteries can be attached later to power the device.

Solution should be able to detect something going in front of it

Sensing something going by can be accomplished in a number of ways. Existing methods utilize inductive loops, IR, piezoelectric strips, road tubes, and even video in some cases. The solution should implement one of these to detect an object passing by.

Solution should be able to tell the difference between a bike and other forms of travel

When the sensor is triggered, then the solution should have some way to compare the new data to some data that allows the sensor to determine if a bicycle has passed it, and not a motorcycle, car, pedestrian, or other mode of transport.

Solution should allow for manual retrieval of data

In the event wireless connectivity is knocked out, then there must be a way for the device to be able to save images to a retrievable medium. This way, data can still be collected until Internet service is restored.

Solution should allow for remote transmission of data

Transmitting data is essential for allowing the sensor to remain low cost and low power. If data can be transmitted, then more powerful machines can process data gathered from the sensor. Potential methods would include 3G, Wi-Fi, Bluetooth, or other wireless transmissions.

4.2 System Requirements

System will use a low-cost microcontroller as the main controlling device

A microcontroller is a device that has a CPU, RAM, and some programmable memory to take inputs and produce desired outputs. The Particle Photon is a microcontroller that is easily programmable, with many

libraries available to it to use additional sensors and attachable peripherals. The Photon is available for $19 [13].

Solution will use a low power microcontroller to facilitate communications between serial peripherals

The Particle Photon is a microcontroller similar to an Arduino that utilizes a much more powerful ARM processor. It also has a free development environment similar to Arduino. The Photon is capable of serial communications with several devices. This is done though several digital and analog pins on the device. Using these pins, peripheral devices like sensors or storage can be added to the Photon.

Solution will have local storage

Adafruit sells a breakout board that a user can insert a microSD card into to store data. MicroSD cards can be purchased easily and cheaply, and software libraries exist for the Photon to use SD cards as a storage medium. The breakout board is available for $7.50 at the time this document was created [14].

Solution will use peripherals capable of serial communications

Adafruit sells a weatherproof VC0706 serial camera on their website. The image sensor is capable of being interfaced with the Particle Photon after modifying the Adafruit software libraries that control the camera from a microcontroller, specifically an Arduino. Adafruit libraries can be ported to the Particle platform by replacing some code to use the Particle environment instead of the Adafruit and Arduino environment. The camera is available for $54.95 [15].

The breakout board uses serial to communicate with the device as well. Using these devices, serial communications can take place with the microcontroller, allowing for the sensor to function as intended.

Solution will have manually retrievable data

Storing data on an SD card makes it easy to manually retrieve the data. Swapping the SD card can happen quickly, and the small format with large volumes can store many readings from a sensor.

Solution will be capable of remote transmission of data

The Particle Photon utilizes an on-board Wi-Fi chip, capable of 2.4 GHz 802.11b/g/n communications. The microcontroller can communicate with remote servers using existing libraries for the Particle Photon, and data can be sent to the Internet via TCP and IP protocols.

Solution will use a sensor to detect a passing object

The VC0706 camera sold by Adafruit has motion detection built in, and is controllable via modified libraries that will work with the Particle Photon. This is done by detecting a change in the current image that the camera is capturing. In other words, if something large enough moves into the camera's frame, then it knows that motion has been detected.

Solution will accurately identify objects passing in front of sensor

TensorFlow is a machine learning and deep learning algorithm developed by Google. It is an open source project, and is actively being developed. Machine learning is capable of classifying images and identifying objects in the image. Using the Adafruit VC0706 image sensor with the Particle Photon will let the solution send data to the remote server with the TensorFlow program on it.

The solution that was built in this project took these requirements and, using a Particle Photon, outdoor serial JPEG camera, SD card breakout board and SD card, constructed a functioning prototype. The sensor unit was then programmed to send image data to a remote server containing a machine learning

algorithm called TensorFlow. TensorFlow was trained to classify images for this project based of images returned from Google image searches.

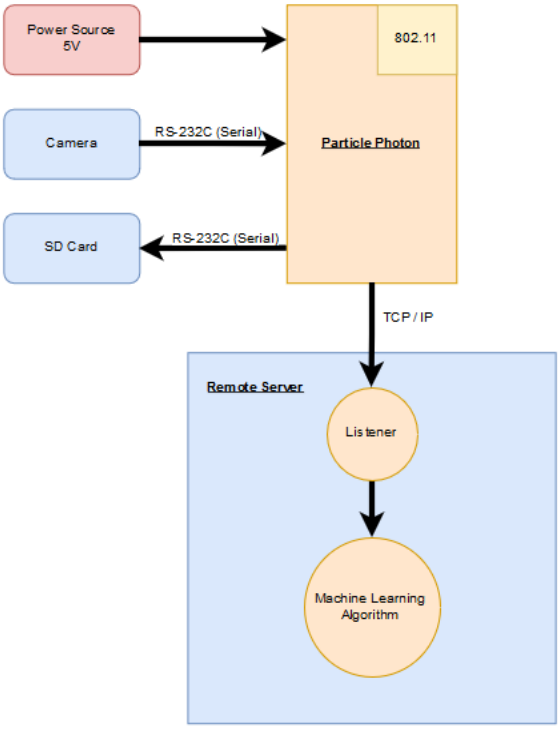


Figure 12 System Diagram with Proposed Hardware

4.3 Hardware and Physical Wiring

The bike sensor was constructed using a Particle Photon, an Adafruit VC0706 Weatherproof camera, an Adafruit SD breakout board, and a microSD card. Wiring is based off of the Adafruit tutorial [ ] for the VC0706 [16] sensor. Wiring of the sensor is detailed in Figure 13.

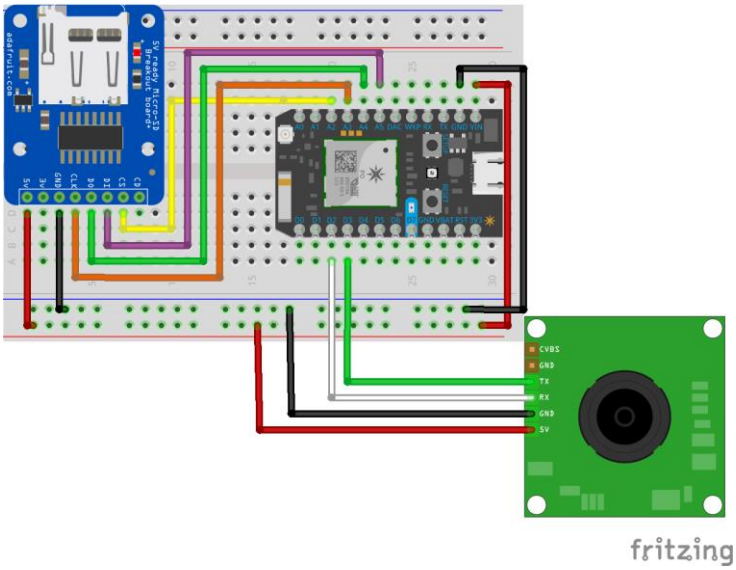


Figure 13 Bike Sensor Wiring

The final sensor was constructed using the Ohio University ITS computer labs, facilitating soldering, testing, and providing a VM for the TensorFlow instance. Using the following table of items, the sensor was built and programmed using the Particle Dev programming environment installed on the ITS lab computers. The final sensor is represented in the following figure.

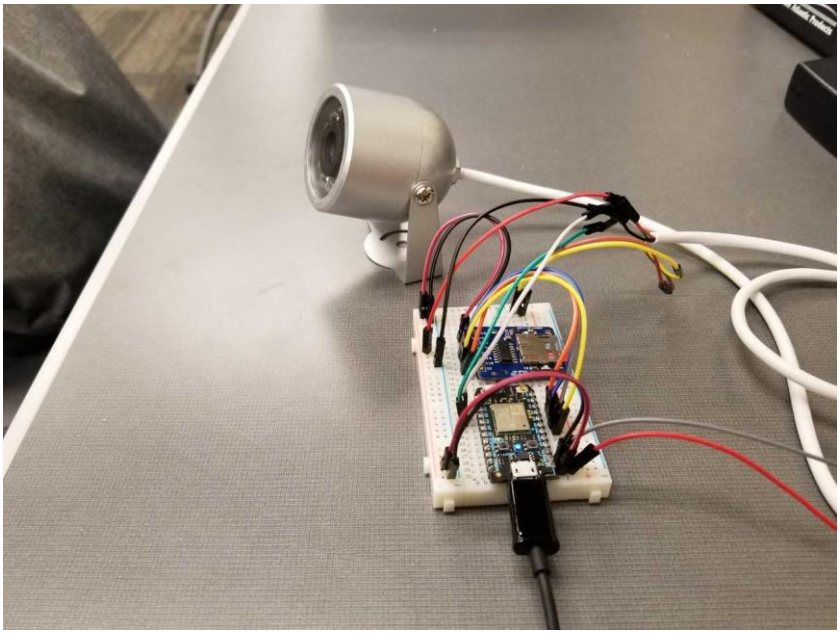
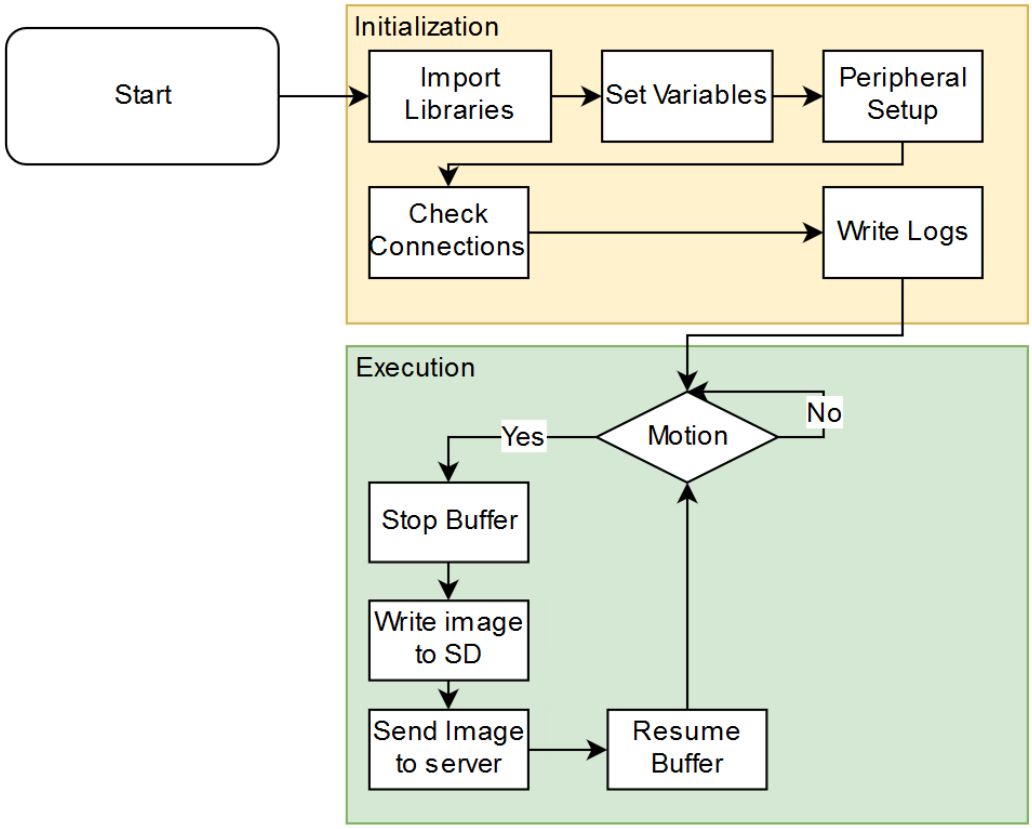


Figure 14 Final Sensor Construction

4.4 Software

The software that is used on the Photon has two distinct sections. The *setup()* function acts as an initialization point for the sensor. It starts a variety of functions and creates variables that will be used for the program. The *loop()* function is the main execution of the program. Motion is detected, images are saved and sent, and the camera continues looking for motion. The specific function of the code is detailed below. A diagram of the code is in Figure 12. The diagram details a high-level explanation of what the code does in its two main functions, labeled as *Initialization* and *Execution* for the *setup()* and *loop()* functions respectively. More complex functions can be added to the code, but are considered out of scope for this project.



Code for the sensor is located at the OUSmartInfrastructure Github page [16]. It contains the uncompiled code used to create firmware that is uploaded to the Particle Photon to facilitate communications between peripherals and the remote server. An unedited version of the code is available in the Appendix of this document. More details are available in the corresponding section for accessing the code via GitHub.

The required libraries include the modified Adafruit_VC0706 library. This includes the changes made to use the same code on the Particle platform. The main changes that needed to be made were replacing Arduino specific libraries with the libraries compatible for the Particle platform. The code begins with importing several required libraries:

```
#include <Adafruit_VC0706.h>
#include <SPI.h>
#include <SdFat.h>
#include <ParticleSoftSerial.h>
```

The *Adafruit_VC0706* library used in this project has been customized for the Particle platform. Importing the SPI (Serial Peripheral Interface) library lets the SPI configuration of the Photon device be set for communications with serial devices attached to the Photon. This is required for the SD card, which uses the SdFat Library. The final library, ParticleSerialSoft mimics the SerialSoft and NewSoft libraries for Arduino, which are used as a way to use digital communications on other pins, and not just the designated TX and RX pins on an Arduino. This is required for communications to the camera.

The next sections set up variables that will be used later, as well as setting the SPI configuration.

```
// Setup SPI configuration
#define SPI_CONFIGURATION 0
// Primary SPI with DMA
// SCK => A3, MISO => A4, MOSI => A5, SS => A2 (default)


//Declarations
ParticleSoftSerial cameraconnection = ParticleSoftSerial(D2, D3);
Adafruit_VC0706 cam = Adafruit_VC0706(&cameraconnection);
```

```
SdFat sd;
const uint8_t chipSelect = SS;
String logFile = "sensor.log";
//cameraDelay = 0;

//TCP Setup
TCPClient client;
byte server[] = {#,#,#,#}; // Local machine
int port = #;
```

The *cameraconnection* variable is used to specify the pins that the camera is on. In the case of this project, digital pins 2 and 3 were used for the camera communications. The modified Adafruit library is used to create a new camera variable using the connection that was set up in the previous line. The *SdFat* library is used to create a new variable *sd* to be used later when writing and reading from the SD card. The *chipSelect* variable is used to specify the Slave Select, which is required to set for proper functionality when using the SD card. The Photon acts as the master, and the SD breakout board is the slave, so setting the *chipSelect* pin to the SS pin of the Photon allows it to communicate with the SD card when the Photon has data to send. The *logFile* string is used to specify the name of the log file that is created later in the code. Commented out is a *cameraDelay* variable. This can be used to 'tune' the device, and capture photos of bikes as they get farther into the frame. Without this set, the camera buffer (an image) is written to the SD card the moment that motion is detected. Often, this is the edge of a tire of a slow-moving object.

The next section sets up variables required for a TCP communication between the particle and a remote server. The *server* variable is a list of 4 numbers that are used by the native TCPClient library to communicate via TCP. The *client* variable is the new variable that will be used to connect to the remote server. The *port* variable specifies the port the remote server is listening on for new image data.

The next part of the code is the ***setup()*** function on the device. Its function is to initialize all variables and check for proper setup of the device. It checks for the SD card, and attempts to connect to the camera.

```
void setup() {
    File myFile;
    pinMode(chipSelect, OUTPUT); // SS on Photon
    SPI.begin();
```

```
    Serial.begin(9600);
```

This section creates a new variable called *myFile* that is used to refer to a log file that is being written to. This section also sets the chipSelect pin to be an output of the device. This allows the SdFat library to set the state of the pin, and not have the Photon 'listen' to the state. SPI communications can be started after the third line, so now the device can be told about the SD card as well as the camera that are attached to the Photon. Serial communications begin for debug and error reporting on a local machine.

```
    while (!Serial) {
        SysCall::yield();
    }
//Press a button to start setup
// This is being used for debug now, but will be disabled or removed
    while (Serial.read() <= 0) {
        SysCall::yield();
    }
    delay(1000);
```

This section simply waits for any input from the user over a serial connection before moving on. This was added for debugging purposes, and can be commented out for use while deploying the sensor.

```
//-----------------------------------------------------------------------
    //Added extra cam.begin() to allow the second one to return true
    //Unsure why this is needed (3/12/18)
    cam.begin();
```

Calling *cam.begin()* is needed because the first call always would return a zero during sensor construction. However, additional calls to the same function did not return a zero after the method was called once. This was left in the code to ensure that the device would function properly. Even if it returns a one, then the camera should still function as intended.

```
//Try to locate the camera
        if (cam.begin()) {
          Serial.println("Camera Found:");
        } else {
          Serial.println("No camera found?");
          //return;
        }
```

This section makes an attempt to detect the camera by starting communications with it. The *cam.begin()* function starts the camera communications over serial. In this case, we have told the photon to treat digital pins 2 and 3 as serial communications pins, which allow the Photon to communicate with it and the SD card at the same time.

```
 //Check for SD card
delay(1000);
Serial.println("Initializing SD Card");
if (!sd.begin(chipSelect, SPI_FULL_SPEED)) {
    sd.initErrorHalt();
}
delay(1000);
```

This section tries to detect the SD card by starting serial communications. The *chipSelect* variable is used to tell the SD card when it is being written to or being read from. The *SPI_FULL_SPEED* variable is used to tell the SD library that he maximum possible SPI speed should be used for SD card communications. If this communication fails at this point, then an error is printed to the serial console. Once this is done, a delay is put in before setting some settings on the camera.

```
cam.setImageSize(VC0706_640x480);        // biggest
cam.setMotionDetect(true);           // turn it on
cam.setCompression(99);
```

This section sets the camera's image resolution, enables motion detection, and sets the compression ration to the near max. The higher resolution is needed for the TensorFlow installation, where a larger image will make it easier for TensorFlow to determine what is in the image. The compression was set so high to allow for smaller file sizes, which will increase both SD card write time and TCP transmission time.

```
 //Start write to SD card by opening file
if (!myFile.open(logFile, O_RDWR | O_CREAT | O_AT_END)) {
    sd.errorHalt("opening sensor.log for write failed");
}

//If the file opened okay, write to it:
Serial.println("Writing to sensor.log ...");
myFile.printf("Startup complete: ");
myFile.println(Time.timeStr() + "\n");
```

```
            //Close the file:
            myFile.close();
            Serial.println("Startup complete");
            Serial.println(Time.timeStr());
```

This section opens the log file and writes some startup data to it, specifically when the device was turned on. This could be useful for troubleshooting issues with the device later. If the file cannot be opened, then the log file is not written to. Once the file is written, the file is closed, which saves the written data. User motivation that the startup process completed successfully is printed to the serial console.

```
            Serial.println(WiFi.localIP());
            delay(50);
            //client.connect(server,51234);
}
```

This final section of the *setup()* function is giving the user some more information about the device, which is the IP address that the Photon has received. This can help with troubleshooting, and give a user useful information about communications with the device.

The entire *loop()* function takes place as soon as possible. The sections of code that run are executed if motion is detected. Each section has a distinct function to it.

```
void loop() {
    File myFile;

    if (cam.motionDetected()) {
       Serial.println("Motion!");

//Turn off motion detect to keep buffer for saving image
       cam.setMotionDetect(false);

      //delay(cameraDelay)
      if (! cam.takePicture())
        Serial.println("Failed to snap!");
      else
        Serial.println("Picture taken!");
```

This section is where the camera checks for motion, using the *motionDetected* function of the modified Adafruit_VC0706 library. Serial output is sent to the user. Motion detection is then turned off, to prevent potential overwriting of the current camera buffer if additional motion is detected. The commented-out

delay command uses the previously set variable to delay when the image is taken. The camera stops the buffer when the *takePicture* function is called. A successful or unsuccessful status is printed to the serial console.

```
       char filename[13];

//Currently allows for up to 100 photos to be saved to SD
      strcpy(filename, "IMAGE00.JPG");
      for (int i = 0; i < 100; i++) {
        filename[5] = '0' + i/10;
        filename[6] = '0' + i%10;

//Create if does not exist, do not open existing, write, sync after write
        if (! sd.exists(filename)) {
          break;
        }
      }
```

At this point, the code begins setting up the data needed to write the camera buffer to the SD card. It begins by creating a new *filename* variable that can count between 0 and 99, allowing for 100 unique images. It replaces the numbers at the end of the string “IMAGE00.JPG” to correspond the correct number that the image is taken in.

```
//Print info to log file
      myFile.open(logFile,FILE_WRITE);
      myFile.printf("Picture taken: ");
      myFile.printf(filename);
      myFile.printf(" ");
      myFile.printf(Time.timeStr());
      myFile.println();
      myFile.close();
```

This section prints the time and filename of the image that was just detected. This could be useful for troubleshooting later should the image fail writing or sending.

```
      File imgFile = sd.open(filename, FILE_WRITE);

      uint16_t jpglen = cam.frameLength();
      Serial.print(jpglen, DEC);
      Serial.println(" byte image");

      Serial.print("Writing image to "); Serial.print(filename);
      int32_t time = millis();
```

This section creates a new file on the SD card for writing data called *imgFile*. The code then gets the length of the image in bytes and prints it to the serial output. A timer is started to record how long the writing process takes.

```
 //While there is still data to read...
while (jpglen > 0) {

   // read 96 bytes at a time;
  uint8_t *buffer;
  uint8_t bytesToRead = min(96, jpglen);
  buffer = cam.readPicture(bytesToRead);
  //Write the image
  imgFile.write(buffer, bytesToRead);
  jpglen -= bytesToRead;
}

imgFile.close();
Serial.println("...Done!");
```

This section is where the image is written to the SD card. A temporary buffer of data is created. The next number of bytes are calculated, up to a maximum of 96 bytes. The *buffer* variable is assigned to the next bytes that are returned from the *readPicture* function. The *buffer* variable now contains up to 96 bytes of data, which are then written to the end of the image file. This continues if there is more data to read from the image. Once the loop completes, the image is saved and closed.

```
Serial.println("Sending file to server...");

myFile.open(filename, O_READ);
int data;
long int now = millis();
if (client.connect(server,port)){
    Serial.println("Connected!");
    while ((data = myFile.read()) >= 0){
        client.write(data);
    }
    client.stop();
}
```

This section is where the image is sent to the remote server. The file where the image was written is opened for reading. Using the *client, server,* and *port* variable, the code checks for a successful connection to the remote server and reports a success to the serial console. The file is then gathered into chunks and

send over TCP to the remote server by calling the *write* function from the *client*. Once the file is complete, the connection is closed.

```
            long int sendTime = millis() - now;

            Serial.printlnf("Time to write was: %d",sendTime);
            //Particle.publish("Picture-Taken",filename);


            time = millis() - time;
            client.println(time);
            Serial.println("done!");
            Serial.print(time); Serial.println(" ms elapsed");
            cam.resumeVideo();
            cam.setMotionDetect(true);
           }
      }
```

This final section of code calculates how long it took to send the file to the server and the total time for all actions to take place. This could be recorded for troubleshooting information, but it is not in this case. The camera resumes and allows the current buffer to be overwritten and allows for new motion detection to be detected. After this point, the entirety of the *loop* function is run again to check for new motion and write and send a new image.

Code that is placed on the server is attached in the appendix of this report.

4.5 Accessing Information on GitHub

GitHub is an online repository for computer code. GitHub hosts code and allows for creators to manage version control and access control while providing ways for new contributors to update code, flag bugs, or request features.

All code in this project is hosted on the OUSmartInfrastructure GitHub repository for this project. It can be accessed by the cited link, and the entire code package can be downloaded from the webpage. An Ubuntu 64-bit server was used for this project, and had no GUI to navigate to the webpage. By using the

git command on an Ubuntu server, a copy of the code can be downloaded. Once downloaded, the specific programs will need to be moved to their proper folder.

If a researcher wants to continue development, then downloading the folder titled *BikeSensorv3* is the folder that contains all source code for compiling in a separate environment. The *lib* folder contains all libraries used in the project, and the *src* folder contains the source code for the *.ino* file. With these two folders, a user can import the project into the Particle Dev environment for additional development. Even if the machine learning component is not used, the sensor will still function as a device for manual retrieval or to simply send a file to a server.

The other programs in the repository are python scripts that make functions easier. The *startServer.py* program serves as the listener for the device. The Photon is coded to send an image to a specified web address where it is intended that some mechanism of getting the image is possible. This startServer.py contains a way to check that the images are jpg files, and saves them accordingly. It is intended to be placed inside of the TensorFlow instance folders, where a custom folder for to-be-classified images are located. Structure is up to the researcher to place, but code for this project dictates it must go inside of the TensorFlow_for_Poets folder and be run from there.

The *startClassify* python script acts as a way to process a batch of images to classify. Each image is fed into the machine learning algorithm and the results are printed at the end of the program, as well as saved to a *.tsv* file. This script is not integral to the function of the sensor, but makes classifying a large number of images easier, and allows for the possibility of automatically classifying images. This program is intended to be placed in the same location as the previous one, which is just inside the Tensorflow_for_Poets folder.

The main page of the repository will look like the image below. It contains 2 programs and a folder that have required files to run the sensor as it functions in this project. The top arrow contains the source code, libraries, and wiring diagram. The other two arrows are the python scripts for gathering the jpg images on the server and the bulk classification script.

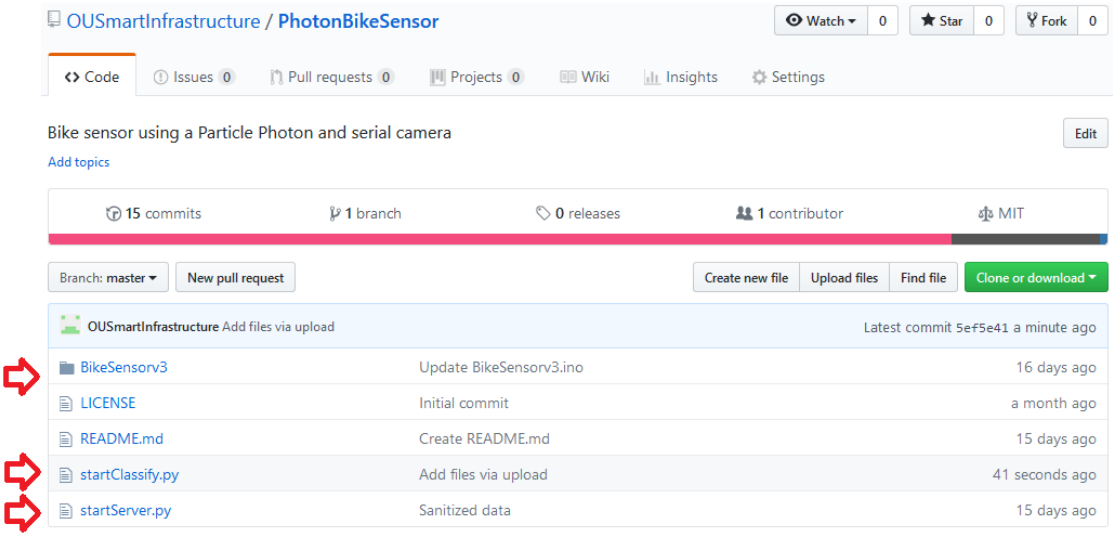


Figure 15 GitHub Repository Project Page

Inside the BikeSensorv3 folder will look like the following image.

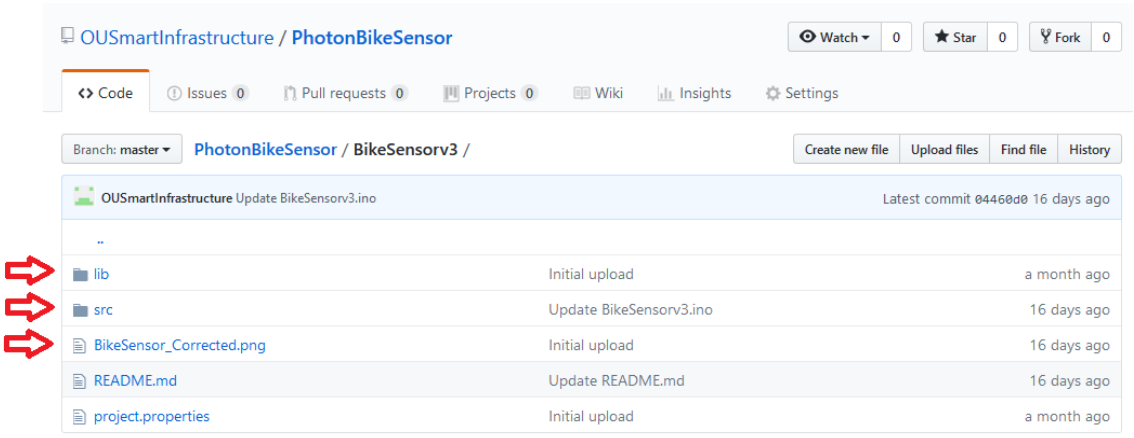


Figure 16 Project Libraries, Code, and Wiring

The top arrow contains the libraries used in this project. The second arrow is the actual source code for the project. By placing these two folders into your Particle development environment, firmware can be flashed to a device. The last arrow contains a Fritzing diagram of how to wire the sensor with the current code.

5. TensorFlow and Machine Learning Algorithms

Machine learning uses statistical and mathematical algorithms take input data and then compare to training data. How the input data relates to the output data is determined by the algorithm and it delivers an output.

Installation of TensorFlow was followed using the instructions located on the code labs pages from Google [18]. This instance used the native *pip* installation and created a folder on the server with the required files automatically. Getting code using the *git* command is detailed on the CodeLabs page as well.

In this project, machine learning is applied to image classification. A classifier is an algorithm that takes input data and identifies a category that the item belongs to. For example, images of a sailboat and a tugboat may fall into the category of boats. The algorithm is given data to train itself to identify new images. The training takes place when a human gathers information to present to the machine learning algorithm and tells the algorithm what the data is, and what the correct output should be. This is called feature extraction, and can easily be done by humans, but is difficult for machines to do on their own. A machine is capable of doing this, and it is called Deep Learning.

The instance of TensorFlow that is used is called *TensorFlow for Poets*, which comes with a pretrained classifier called Inception. It also has prewritten programs to classify images and retrain Inception, among other. This runs using the Python programming language.

In this project, organized images are presented for training to the algorithm. The machine is then able to compare the human's extracted features and breaks an image into smaller pieces to extract detailed information about the image. Now, the training data that will be the basis of what this algorithm knows. It should be said that the images should match the category that the image is in; for example, the data for cars should only contain images of cars, and not any images of flowers. The machine learning algorithm will learn how these images differ when placed into the proper categories, and be able to identify them accordingly.

TensorFlow is an open source library for numerical computation, specializing in machine learning [17]. The CodeLabs pages cited contains an instructional tutorial for running TensorFlow and training a classifier to identify images. This tutorial was largely the basis for completing the machine learning portion of this

project [18]. Custom software was written to make certain functions easier to do, and is included on the OUSmartInfrastructure GitHub repository for the project.

The TensorFlow instance used for this project was trained to identify 4 different categories: people, bikes, people on bikes, and cars. These 4 categories were chosen for the purpose of showing differentiation between a single person, a vehicle, and a bicycle. A person on a bicycle was added to determine if the algorithm could tell the difference between a person, and a bicycle, and understand that a person on a bike is different from the two individual things that make it up. It is also the focus of this sensor, and the machine output could be used to count the number of bicycles riding by the sensor. Training data consisted of images found from Google Image searches for each corresponding category. About 60 images for each category were used. Training is done by using the commands listed in the TensorFlow tutorial.

The program *startClassify.py* starts off classifying a batch of images inside a specified folder. This program is also the mechanism for getting output from the device. This program counts the identified objects from images currently in the testing folder.

The heavy lifting of actual classification is taken care of by the included programs with the TensorFlow instance. All a user needs to do is specify images to classify and execute the code. The *startClassify* script was made to make processing a batch of images easier, and recording all the results in a single place.

6. Experiment Design

To test the sensor, an experiment was conducted. The purpose of the experiment was to test both the functionality of the sensor in a variety of conditions, as well as the successful classification of images captured and sent to the TensorFlow algorithm. Testing the sensor in a controlled way indicates how well

the sensor can detect bicycles passing by at predetermines speeds, angles, and times of day. The TensorFlow algorithm should also be able to differentiate bicycles from other objects at the same speeds, angles, and times of day. These were the only factors that were controlled, along with the same researcher and same bicycle being used in the same test at the same physical location. One factor that was not controlled was distance from the sensor.

All experiments followed the same general pattern, using the researcher as the test subject. The figure below shows an example of tests that were conducted from a top-down view. Arrows indicate possible directions the researcher could travel for a valid test. The rectangle represents the researcher, and the oval and circle represent the sensor that was constructed for this project.

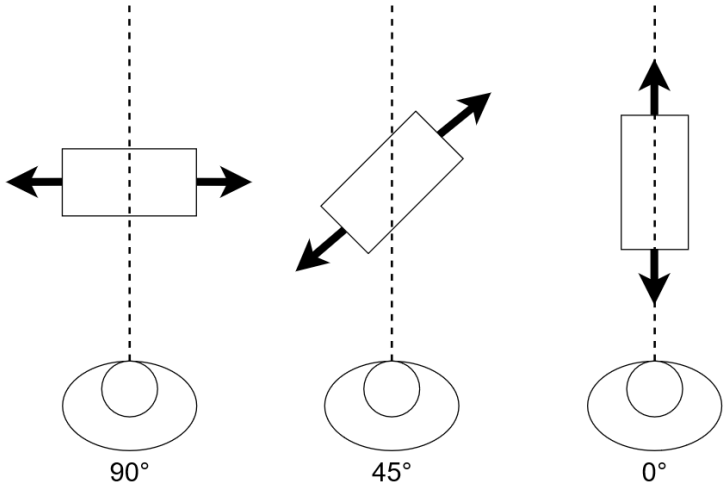


Figure 17 Diagram of Angles Tested

The sensor was activated via a laptop USB port to power the device and allowed the device to operate normally, while the researcher rode a bicycle by the sensor. If the sensor successfully detected movement during the time the researcher rode by, then an image should be captured. The success of these detections is detailed in the Results section of this document. Once images were captured, they were uploaded to the machine containing the TensorFlow instance using the manual retrieval method, since Wi-Fi was

unavailable at the testing location. The images were classified by the TensorFlow instance and the results were recorded, also detailed in the Results section of this document.

Table 1 Experimental Factors and Levels to be Tested

Factor	Level					
Time of Day	Day			Night		
Bike Speed (mph)	5	10	15	5	10	15
Angle (degrees)	90	45	0	90	45	0

The selected factors and levels were chosen because of the difficulty a machine learning algorithm may have in identifying bicycles at a variety of angles, speeds, and at different times of day. Realistically, travelers may not always go by the sensor to perfectly match the expected orientation each time. By providing flexibility by way of training the algorithm at a variety of camera angles, the sensor can be placed in any convenient way in a target area and still be able to accurately count bicycles.

7. Results

7.1 Sensor Detection

Table 1 contains the variables that were tested and their results. In table 1, a successful detection was labelled with a 1 and highlighted green. Unsuccessful detections were labelled with a 0 and highlighted red. Three trials were run for each speed at each angle at each time of day.

Table 2 Detection Experimental Factors Results

Angle: 0 degrees							
Speed (Mph)		Day			Night		
		1	2	3	1	2	3
	5	1	1	1	1	1	1
	10	1	1	1	1	1	1
	15	1	1	1	1	1	1

Angle: 45 degrees							
Speed (Mph)		Day			Night		
		1	2	3	1	2	3
	5	1	1	1	1	1	1
	10	1	1	1	1	0	0
	15	1	1	1	1	1	1

Angle: 90 degrees							
Speed (Mph)		Day			Night		
		1	2	3	1	2	3
	5	1	1	1	1	1	0
	10	1	1	1	1	1	1
	15	1	1	1	1	1	1

Table 2 contains the successful detections of the sensor. A successful detection is when the camera determines that there is motion in the camera, and an image is written. Table 2 details all test cases, resulting in 54 possible cases. Of 54 possible detections, 51 successfully wrote an image.

The experiment consisted of the researcher travelling by the sensor at the specified speeds, at the specified angle, during the specified time of day. All detections were successful, except for trials 2 and 3 at 10 mph, and trial 3 at 5 mph. This could be because of the distance from the sensor was not factored in while conducting the test; while the researcher may have passed by the sensor, they may have been too far away to trigger a detection, especially at night where visibility is reduced. The researcher wore dark clothing during all tests, and this may have contributed to the unsuccessful detections. Regardless, the sensor has a detection rate of 94.4%.

7.2 Machine Learning Classifications

Table 2 contains the results of the machine learning classifications. Each row contains the time of day, speed, and angle of the test, as well as the best classification returned by the TensorFlow instance and its corresponding score. Only the top classification was gathered, since these would be the best guess from the TensorFlow instance as to what is in the image based on training data that was used.

Table 3 Raw Data Gathered from Testing and TensorFlow Algorithm

ToD	Speed	Angle	Classification	Score	ToD	Speed	Angle	Classification	Score
day	5	0	personbike	0.608231	night	5	0	personbike	0.491342
day	5	0	personbike	0.903196	night	5	0	personbike	0.75013
day	5	0	personbike	0.930174	night	5	0	personbike	0.633497
day	5	45	personbike	0.984515	night	5	45	cars	0.997846
day	5	45	personbike	0.729837	night	5	45	cars	0.997091
day	5	45	personbike	0.742719	night	5	45	cars	0.998774
day	5	90	personbike	0.797225	night	5	90	cars	0.992149
day	5	90	personbike	0.775284	night	5	90	cars	0.985469
day	5	90	personbike	0.881409					
day	10	0	personbike	0.775444	night	10	0	cars	0.990467
day	10	0	personbike	0.917822	night	10	0	cars	0.976116
day	10	0	personbike	0.969311	night	10	0	cars	0.956472
day	10	45	personbike	0.740486	night	10	45	cars	0.994405
day	10	45	cars	0.543542					
day	10	45	personbike	0.89698					
day	10	90	personbike	0.961405	night	10	90	cars	0.99834
day	10	90	personbike	0.625195	night	10	90	cars	0.995846
day	10	90	personbike	0.666215	night	10	90	cars	0.996145
day	15	0	personbike	0.690353	night	15	0	cars	0.667743
day	15	0	personbike	0.937485	night	15	0	personbike	0.50356
day	15	0	cars	0.991296	night	15	0	cars	0.863367
day	15	45	personbike	0.617153	night	15	45	cars	0.900785
day	15	45	personbike	0.969995	night	15	45	cars	0.726965
day	15	45	cars	0.765346	night	15	45	cars	0.942226
day	15	90	cars	0.644794	night	15	90	cars	0.978169
day	15	90	personbike	0.540687	night	15	90	cars	0.960071
day	15	90	cars	0.848758	night	15	90	cars	0.974016

This table contains the data that was gathered from the experiment that as designed for testing this sensor. When detections were not able to be recorded, a blank line was inserted into the table, signifying that the researcher rode by the sensor, but the sensor was unable to detect motion going by the sensor. Viewing this table, it is observed that the sensor was able to detect motion in almost all cases, namely night testing being the only failed detections. It can also be observed that the daytime

tests were largely successful at correctly classifying the image, while the nighttime classifications were unsuccessful and often returned a misclassified result from the TensorFlow instance. The following table contains successful machine learning classifications in a similar format to the detections listed in Table 2.

Table 4 Successful Machine Learning Classifications

Angle: 0 degrees							
Speed (Mph)		Day			Night		
		1	2	3	1	2	3
	5	1	1	1	1	1	1
	10	1	1	1	0	0	0
	15	1	1	0	0	1	0

Angle: 45 degrees							
Speed (Mph)		Day			Night		
		1	2	3	1	2	3
	5	1	1	1	0	0	0
	10	1	1	0	0		
	15	1	1	0	0	0	0

Angle: 90 degrees							
Speed (Mph)		Day			Night		
		1	2	3	1	2	3
	5	1	1	1	0	0	
	10	1	1	1	0	0	0
	15	0	1	0	0	0	0

Table 4 details the successful classifications from the trained machine learning algorithm. The sensor did relatively well during the day, with 5 misclassifications out of 27, or an 81.4% success rate of correctly classifying a biker during the day. This is comparable to the studies that were conducted by Nordback. However, at night, the sensor did significantly worse. The sensor only classified 4 of of 24 images correctly. This is a success rate of 16.7%. There are several potential sources of error in this case, discusses in the next section.

7.3 Power

Power consumption was a major goal of this project. The Particle Photon was selected for its abilities as a microcontroller, as well as its low power features and low cost. Power measurements were gathered while the device operated normally through different states. The observed states were the *Idle* state, where the device was only running background operations and powering peripheral devices. The *Saving* state occurs after an object is detected and the Photon is writing the image to the SD card. The final *Sending* state was observed when an image is finished being written, and is sent over the internet using the on-board Wi-Fi chip. The following table contains 3 measurements of each state and the average for those three measurements.

Table 5 Completed Device Power Consumption

State	Values (Amps)			Avg
Idle	0.15	0.16	0.15	0.1533
Saving	0.16	0.16	0.16	0.1600
Sending	0.18	0.17	0.17	0.1733

The device used an average of 0.1622 Amps or 162.2 mA. This is low enough to be powered from a battery or solar panel for an extended period of time, depending on battery size. Assuming optimal conditions, the device could run off a 2500 mAh battery for almost 11 hours during the day.

However, the model of camera used had several infrared LEDs for night time visibility. A measurement of the camera power consumption was conducted as well, and the results are recorded in the table below.

Table 6 IR Camera Power Consumption

State	Values (Amps)			Avg
IR Off	0.08	0.08	0.08	0.0800
IR On	0.28	0.28	0.28	0.2800

The table shows that the power consumption increases significantly when using the camera's IR LEDs. The camera alone uses 280 mA when the LEDs are activated. This increases the nighttime power consumption of the device to be an average of 362.2 mA during night time usage. The same sized battery would last less than 5 hours if used to power the sensor at night. A larger battery could be used in this case, or power can be reduced some other way. However, this is still able to be powered by current Li-Po batteries.

7.5 Enclosure Design

A 3d printable enclosure was designed for this sensor. It is designed to fit the current prototype as is for real world deployment. It is assumed that the camera will mount to some other point and will not need protection as it is weatherproof. The main function of the case is to provide protection for the delicate electronics. The following figure contains a render of the final design for the case to house the electronics.

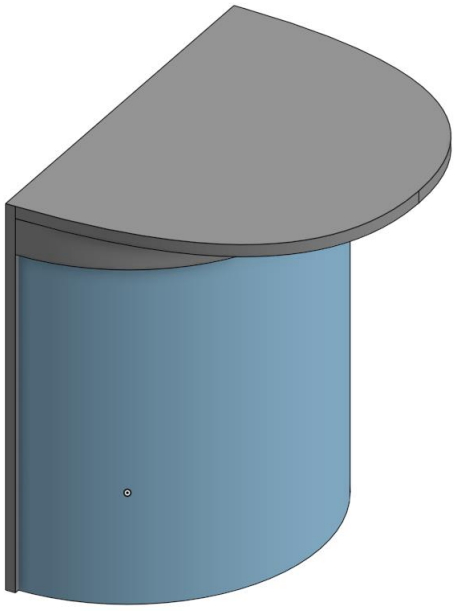


Figure 18 Enclosure Render Together Top

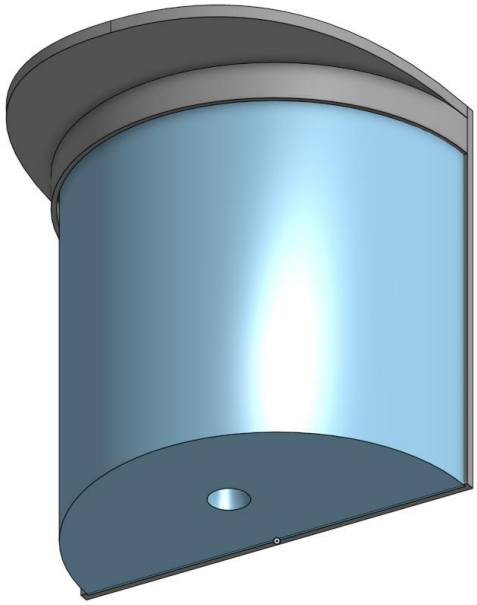


Figure 19 Enclosure Render Together Bottom

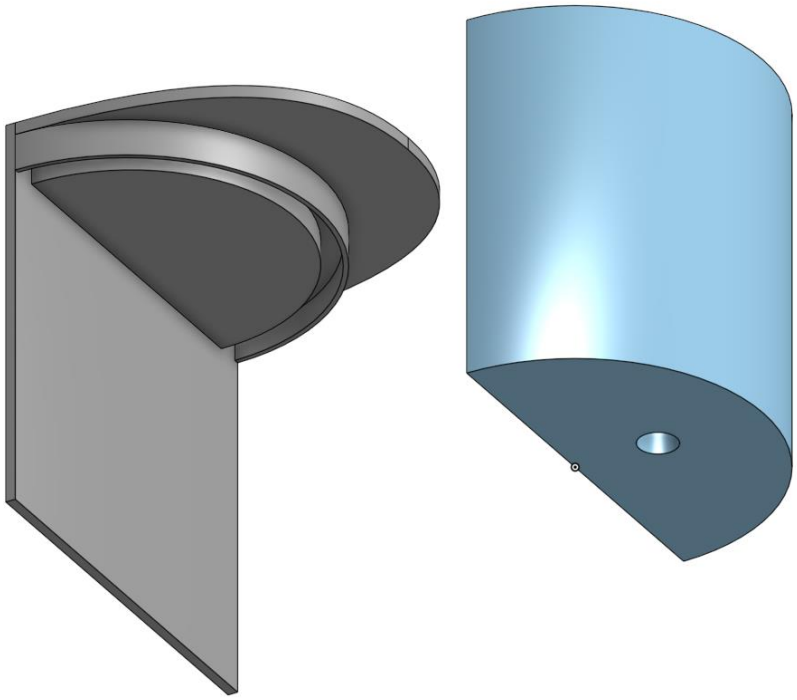


Figure 20 Enclosure Render Exploded Bottom

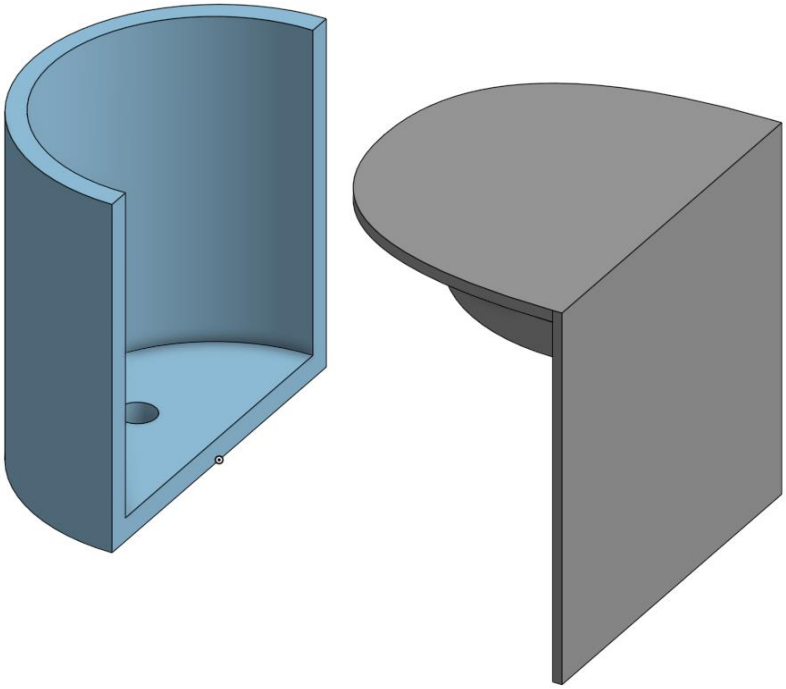


Figure 21 Enclosure Render Exploded Bottom

The sensor was designed to simply enclose all electronics and provide some defense to rain. The enclosure has a curved chamber to house electronics. It only has an outer wall and bottom. The lid has a top and back attached to it. When the two pieces are attached, water should be able to be repelled by design, but the case could also be sealed with silicone or something else to block water from the sides. The curved chamber has a hole in the bottom to accommodate the cord for the camera, also intended to be sealed with silicone. The lid is intended to be a friction fit, and keep water from dripping in the top.

This case may not be ideal, but it provides a way to deploy a sensor and attempts to protect against water. The camera unit could be mounted to the top of the enclosure and the entire enclosure can be mounted to a post or pole and begin to gather data.

8. Conclusion and Future Work

The sensor works for collecting image data. The camera has built in motion detection, and white balance, as well as adjustable focus. These features enable it to be used for a variety of purposes. The sensor takes an image of the object passing near it and sends it to a data server of the Internet, as well as saving it to the SD card. Manual retrieval of the data is possible by retrieving the SD card from the device and move the files to the data server. Regardless of the method used to transfer the data, images are then able to be processed to by the machine learning algorithm present on the data server. The device also uses less than 200mA for each of the actions it takes during the day. This is acceptably low, since the device can be powered by battery and solar (or other alternative sources), for continuous operation. The simple classifier then determines the main subject of the image. The testing confirmed that the sensor did this and reached the goals of the proposed goals.

This project still has several limitations and assumptions.

It is assumed that since users are in public, the images do not need consent to be taken. Privacy in public is considered unreasonable, and thus taking the images as well as transporting the images over plain TCP is acceptable. Privacy concerns are out of scope of this project.

It is also assumed that Wi-Fi is available in a location that the sensor is deployed. This is assumed for this prototype, since the Particle Photon uses Wi-Fi to communicate, and Wi-Fi does not depend on the amount of data sent.

However, several opportunities for additional work exist.

One limitation of the system lies in the training data that was used to train the classifier. As the data shows, night tests performed very poorly. This may be because the testing data did not include any or very few images of people on bikes at night, or even with dark backgrounds. Most images used to train TensorFlow were images of most objects during the day. Even more granular detail, such as training the algorithm to understand a person on a bike can exist with cars, or with people, may be needed. Currently, most of the test images have the only subject of the image as the intended category for that image. For example, the main (or only) subject of an image for a person includes only a single or few people with no additional bikes or cars around it.

Another limitation exists where the sensor did not detect every single instance where a test was run, specifically the night tests. This could be because of a dark setting in the case of night images, or problems with the range of the camera, which were not accounted for in testing. The camera could also have an effective range that was not accounted for during testing. There could also be a problem with how the image is saved, which takes between 8-12 seconds depending on the size of the image. Additionally, sending the image takes time, and both operations are blocking, not allowing new images to be taken. Determining these factors can improve the sensor functionality.

The power consumption can also be improved significantly. The device powers on and automatically powers on the Wi-Fi communications, which greatly increases the overall power usage. Turning the Wi-Fi chip off until a communication is needed would greatly improve the power consumption, and increase potential battery life. For the night time detections, a different method of detecting bikes in the dark may be needed. If a bicycler is not seen at night by the sensor (or a car) it could lead to more significant

problems later. A future researcher may find a more efficient way to power the nighttime functions of the sensor by using a differing detection method.

A measure of the image classifier would be beneficial. A classifier takes a set of data and divides it up to generate a learning set and a training set. In this case, testing with some images that can be easily labeled by a human and presenting them to the classifier could give a baseline measure of how the classifier performs. Since the exact reasoning behind a classifier's decision may not be clear, presenting images similar to those that failed could yield important results for how to improve the algorithm.

This project was aimed at giving cities a means of measuring bicycle traffic for a much lower cost than typically seen in the market. The hardware listed above is available to be purchased for about $100. The cost of running a VM is available from many commercial providers for about $80 per month with 4 CPUs and 8 GB of RAM. This would bring the total monthly cost of the sensor to the cost of the VM to run the machine learning algorithm after constructing sensors. Realistically, one machine can handle many sensors' data as well.

The safety of citizens is always a concern, and the needs of those who depend on safe roadways for bicycle travel should have appropriate infrastructure to accommodate their travel. Cities in turn need appropriate ways to gather data about bicycle travelers in order to provide for their safety. While improvements can still be made, the prototype device functions as intended, and could be viable for cities to use when examining how to measure bike traffic on their roads.

Works Cited

[1] DataUSA, "Athens, Ohio," 2018. [Online]. Available: https://datausa.io/profile/geo/athens-oh/. [Accessed 2018].

[2] Buckeye Hills Regional Council: Regional Transportation Planning Organization, "Long Range Plan," June 2015. [Online]. Available: http://rtpo.buckeyehills.org/long-range-plan/. [Accessed 2018].

[3] Federal Highway Administration, "Safety Benefits of Walkways, Sidewalks, and Paved Shoulders," 2012. [Online]. Available: https://safety.fhwa.dot.gov/ped_bike/tools_solve/walkways_trifold/.

[4] Pedestrian and Bicycle Information Center, "Pedestrian and Bicyclist Crash Statistics," 2015. [Online]. Available: http://www.pedbikeinfo.org/data/factsheet_crash.cfm. [Accessed 2018].

[5] Eco-Counter, "Cyclist Monitoring," 2018. [Online]. Available: https://www.eco-compteur.com/en/solutions/cyclist-monitoring. [Accessed 2018].

[6] e. a. Krista Nordback, "Using Inductive Loops to Count Bicycles in Mixed Traffic," Journal of Transportation of the Institure of Transportation Engineers, 2011 October. [Online]. Available: http://kevinjkrizek.org/wp-content/uploads/2012/04/EcoCounterJOT1011.pdf . [Accessed 2018].

[7] Eco-Counter, "Missoula County - Missoulda Metropolitan Planning Organization," 28 July 2014. [Online]. Available: ftp://ftp.ci.missoula.mt.us/DEV%20ftp%20files/Transportation/Counts/BP/EcoCounter/PROPOSALS/Eco-Counter%20Proposal%20for%20Bicycle%20and%20Pedestrian%20Counting%20Equipment%202014.pdf . [Accessed 2018].

[8] Jamar Technologies, "Bicycle Counters," 2018. [Online]. Available: https://www.jamartech.com/bicyclecounting.html. [Accessed 2018].

[9] MetroCount, "RidePod BT," 2018. [Online]. Available: https://metrocount.com/products/ridepod-bike-tube-counter/. [Accessed 2018].

[10] MetroCount, "FieldPod Remote Data Services," 2018. [Online]. Available: https://metrocount.com/fieldpod/ . [Accessed 2018].

[11] San Francisco Municpal Transportation Authority, "Bicycle Strategy," April 2013. [Online]. Available: https://www.sfmta.com/sites/default/files/BicycleStrategyFinal-accessible_0.pdf. [Accessed 2018].

[12] e. a. Krista Nordback, "bicycle Counts Using Pneumatic Tubes," 3 May 2016. [Online]. Available: http://onlinepubs.trb.org/onlinepubs/conferences/2016/NATMEC/Nordback-KothuriPPT.pdf . [Accessed 2018].

[13] Particle Retail, "Particle Store," 2018. [Online]. Available: https://store.particle.io/. [Accessed 2018].

[14] Adafruit Industries, "MircoSD card breakout board+," 2018. [Online]. Available: https://www.adafruit.com/product/254. [Accessed 2018].

[15] Adafruit Industries, "Weatherproof TTL Serial JPEG Camera with NTSC Video and IR LEDs," 2018. [Online]. Available: https://www.adafruit.com/product/613. [Accessed 2018].

[16] Adafruit Industries, "Adafruit TTL Serial Camera Learn Module," 4 May 2015. [Online]. Available: https://learn.adafruit.com/ttl-serial-camera/arduino-usage. [Accessed 2018].

[17] Ohio University, "OUSmartInfrastructe Photon Bike Sensor," 2018. [Online]. Available: https://github.com/OUSmartInfrastructure/PhotonBikeSensor. [Accessed 2018].

[18] Google, "TensorFlow for Poets," [Online]. Available: https://codelabs.developers.google.com/codelabs/tensorflow-for-poets/#0. [Accessed 2018].

Appendix

BikeSensorv3.ino

```
#include <Adafruit_VC0706.h>
#include <SPI.h>
#include <SdFat.h>
#include <ParticleSoftSerial.h>

// Setup SPI configuration
#define SPI_CONFIGURATION 0
// Primary SPI with DMA
// SCK => A3, MISO => A4, MOSI => A5, SS => A2 (default)


//Declarations
ParticleSoftSerial cameraconnection = ParticleSoftSerial(D2, D3);
Adafruit_VC0706 cam = Adafruit_VC0706(&cameraconnection);
SdFat sd;
const uint8_t chipSelect = SS;
String logFile = "sensor.log";
//cameraDelay = 0;

//TCP Setup
TCPClient client;
byte server[] = {#,#,#,#}; // Remote Server
int port = #;

void setup() {
    File myFile;
    pinMode(chipSelect, OUTPUT); // SS on Photon
    //SPI.setClockSpeed(4,MHz);
    SPI.begin();

    Serial.begin(9600);

    while (!Serial) {
        SysCall::yield();
    }
//Press a button to start setup
// This is being used for debug now, but will be disabled or removed
    while (Serial.read() <= 0) {
        SysCall::yield();
    }
    delay(1000);

//-----------------------------------------------------------------------------

    //Added extra cam.begin() to allow the second one to return true
    //Unsure why this is needed (3/12/18)
    cam.begin();

//Try to locate the camera
        if (cam.begin()) {
          Serial.println("Camera Found:");
        } else {
          Serial.println("No camera found?");
          //return;
        }

//Check for SD card
        delay(1000);
        Serial.println("Initializing SD Card");
        if (!sd.begin(chipSelect, SPI_FULL_SPEED)) {
            sd.initErrorHalt();
        }
```

```
        delay(1000);
        cam.setImageSize(VC0706_640x480);        // biggest
        cam.setMotionDetect(true);           // turn it on
        cam.setCompression(99);

//Start write to SD card by opening file
        if (!myFile.open(logFile, O_RDWR | O_CREAT | O_AT_END)) {
            sd.errorHalt("opening sensor.log for write failed");
        }

//If the file opened okay, write to it:
        Serial.println("Writing to sensor.log ...");
        myFile.printf("Startup complete: ");
        myFile.println(Time.timeStr() + "\n");

//Close the file:
        myFile.close();
        Serial.println("Startup complete");
        Serial.println(Time.timeStr());

        Serial.println(WiFi.localIP());
        delay(50);
        //client.connect(server,51234);
}


void loop() {
    File myFile;

    if (cam.motionDetected()) {
       Serial.println("Motion!");

//Turn off motion detect to keep buffer for saving image
       cam.setMotionDetect(false);

      //delay(cameraDelay) // To allow subject to move a bit more into frame
      if (! cam.takePicture())
        Serial.println("Failed to snap!");
      else
        Serial.println("Picture taken!");

      char filename[13];

//Currently allows for up to 99 photos to be saved to SD
      strcpy(filename, "IMAGE00.JPG");
      for (int i = 0; i < 100; i++) {
        filename[5] = '0' + i/10;
        filename[6] = '0' + i%10;
//Create if does not exist, do not open existing, write, sync after write
        if (! sd.exists(filename)) {
          break;
        }
      }

//Print info to log file
      myFile.open(logFile,FILE_WRITE);
      myFile.printf("Picture taken: ");
      myFile.printf(filename);
      myFile.printf(" ");
      myFile.printf(Time.timeStr());
      myFile.println();
      myFile.close();

//Create image file on SD card
      File imgFile = sd.open(filename, FILE_WRITE);

//Get size of image and print image info
      uint16_t jpglen = cam.frameLength();
      Serial.print(jpglen, DEC);
      Serial.println(" byte image");
```

```
      Serial.print("Writing image to "); Serial.print(filename);
      int32_t time = millis();

//While there is more data to write...
      while (jpglen > 0) {

// read 96 bytes at a time;
        uint8_t *buffer;
        uint8_t bytesToRead = min(96, jpglen);
        buffer = cam.readPicture(bytesToRead);

//Write the image
        imgFile.write(buffer, bytesToRead);
        jpglen -= bytesToRead;
      }

      imgFile.close();

//Send the image to the remote server
      Serial.println("...Done!");
      Serial.println("Sending file to server...");

//Open file
      myFile.open(filename, O_READ);
      int data;
      long int now = millis();
//When connected, send data to server
      if (client.connect(server,port)){
          Serial.println("Connected!");
          while ((data = myFile.read()) >= 0){
              client.write(data);
          }
          client.stop();
      }

//Print timing information

      long int sendTime = millis() - now;
      Serial.printlnf("Time to write was: %d",sendTime);
      //Particle.publish("Picture-Taken",filename);


      time = millis() - time;
      client.println(time);
      Serial.println("done!");
      Serial.print(time); Serial.println(" ms elapsed");

//Resume checking for motion and allow for new buffers to be written
      cam.resumeVideo();
      cam.setMotionDetect(true);
     }
}
```

startServer.py

```
#!/usr/bin/python

import socket, time, os, sys, copy

#Set up info for Internet communications
ip = "0.0.0.0" #This is the IP of the local machine
port = 54321       #Listening port on local machine

#Socket Setup for up to 2 devices to communicate
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind((ip, port))
s.listen(2)

#print user info for debugging
print "Listening on {}:{}".format(ip,port)

while 1:
          #Accept connection and create a new jpeg file with the current time
         client, address = s.accept()
         file = str(time.time())+".jpg"

          #Print debug info and grab first chunk of data up to 1024 bytes
         print "Connection made from {}:{}".format(address[0],address[1])
         data = client.recv(1024)

         while 1:
                    #Keep getting new data in 1024 byte chunks until done
                 oldData = data
                 data = data + client.recv(1024)
                 if oldData == data:
                         break

         #Get into long hex string to confirm proper file format
         checkData = copy.copy(data)
         checkData = str(data).encode('hex')

          #Optional debug info
          #print ("recieved data is: {}".format(data))

          #Check for jpeg SOI and EOI flags
         if ('ffd8' and 'ffd9') in checkData:
                    #If they exist, write the data
                 with open(file, "w+") as f:
                         f.write(data)

                    #Move the images to the testing folder for TensorFlow
                 os.rename(file,str("testImages/"+str(file)))

          #Print output
                print("Image written")
         else:
                print("Valid image not detected")
```

startClassify.py

```
'''
This is a program meant to be run at midnight each night to bulk classify images an update
the resulting counts into a separate data sheet. This way, data can be processed in batches
so that the process is run reliably
'''

import subprocess, os
from os import walk

imageDirectory = "testImages"
resultFile = "results.tsv"
countFile = "counts.txt"
doneDir = "../done/"
files = []
results = []
finalList = []
d = {}

#Get list of all files in directory

for (dirpath, dirnames, filenames) in walk(imageDirectory):
	files.extend(filenames)

#Organize names alphabetically
files.sort()

#Classify each image individually
for file in files:
	print("Evaluating: {}".format(file))
	image = "--image=testImages/"+file
	command = ['python', 'scripts/label_image.py', image]

	process = subprocess.Popen(['python','scripts/label_image.py',image],
	stdout=subprocess.PIPE)

	out,err = process.communicate()
	results.append(out.replace('\n',' '))


#For each item in results (each evaluation of a picture), split into separate elements in a list

for result in results:
	if type(result) is not list:
		finalList.append(result.split(' '))

#Remove first 6 elements of each result, they relate to timing and are useless
try:
	for i in range(len(finalList)):
		for j in range(6):
			finalList[i].pop(0)
except:
	pass

#Now each item in the list is just the classification for each picture. Element 0 and 1 are the
#predicted category and associated percentage

#Send results to a file to track each evaluation and the highest evaluated category
with open(resultFile, 'w+') as f:
	for result in finalList:
		for file in files:
			if(len(result) >=2):
				f.write(file+ "\t result: " + (str(result[0]+": "+result[1])))
				f.write("\n")

#Get current counts, write results to tsv
with open(countFile) as f:
	lines = [line.replace('\n',"").split('\t') for line in f.readlines()]
```

```
#Get initial values
d = {key:value for (key,value) in lines}

for result in finalList:
        if (len(result) >=2):
                matchName = result[0]
                if matchName in d:
                        #Update count in dictionary
                        d[str(matchName)] = str(int(d[str(matchName)]) + 1)
print d

with open(resultFile,'w+') as f:
        for i in range(len(files)):
                result = str(files[i]+"\t{}\t{}\n".format(finalList[i][0], finalList[i][1]))
                print(result)
                f.write(result+"\n")
```