QuikPark

Submitted by:
Keith Kuchenbrod, kjk173
Harsha Dantuluri, hvd8
Yousef Aljallad, yza4
Liam Mccluskey: lmm459

Team Project Number: 43

Advisor:
Professor Roy Yates

May 11, 2020

Submitted in partial fulfillment of the requirements for senior design project

Electrical and Computer Engineering Department

Rutgers University, Piscataway, NJ 08854

1. Introduction

Our Capstone project, an iOS application called QuikPark, addresses the problems that drivers face when they enter a busy parking lot and cannot easily find an available parking spot. In large busy parking lots, drivers often have to search the entire lot for an open spot which results in wasted time, excess traffic, and frustration. The goal of our project is to give drivers directions to the *optimal parking spot*[1] as soon as they enter a parking lot. In order to achieve this goal, we used Raspberry Pis to capture and send images of parking lots, image recognition to detect cars in parking spots, and an iOS application to provide users with directions and occupancy data in a parking lot.

Most existing solutions currently involve some type of sensor to detect if a vehicle is present on a parking spot or not. These systems include both in-ground sensors and overhead sensors that are commercially available and used today. Many of these systems can be found in large cities and parking decks/garages to help alleviate crowding and traffic.

Although systems such as Tinynode, specializing in in-ground sensors, and Smart Parking, which also produces overhead sensor units, yield effective results there are undeniable drawbacks. First is the overall cost of the units. According to Paradox Engineering, the creators of the Tinynode, a single in-ground sensor can run for $130 alone without even taking into account the cost for their networking infrastructures and communications gateway which together cost $1340 (Total cost for installation on Lot 49 would be $5239). Our solution takes into account these expenses and attempts to minimize costs through shrinking the overall numbers of units needed through using cameras.

Our system QuikPark eliminates the need for any type of sensor and uses only image recognition to determine if a vehicle is in a parking spot or not. Using this type of system, we are able to track parking lot occupancy using anywhere from a single camera to a couple at most, depending on the size of the lot. Our system is also not as intrusive as the competitors, due to the minimal effort needed to install these systems. Our cameras can be simply mounted anywhere that a regular security camera would be mounted and be ready to track occupancy data as needed.

QuikPark is designed to innovate upon current smart parking practices by reducing the overall expenditure needed to implement such systems on a large scale. Our solution keeps into mind both the consumer and the facilities in which our systems can be installed in manners not currently available on the market. Through the use of QuikPark, parking can be made easier, faster, and most importantly more convenient through the simple use of our application for our consumers.

[1] The optimal parking spot is defined as the available parking spot nearest to where the driver wants to exit the parking lot

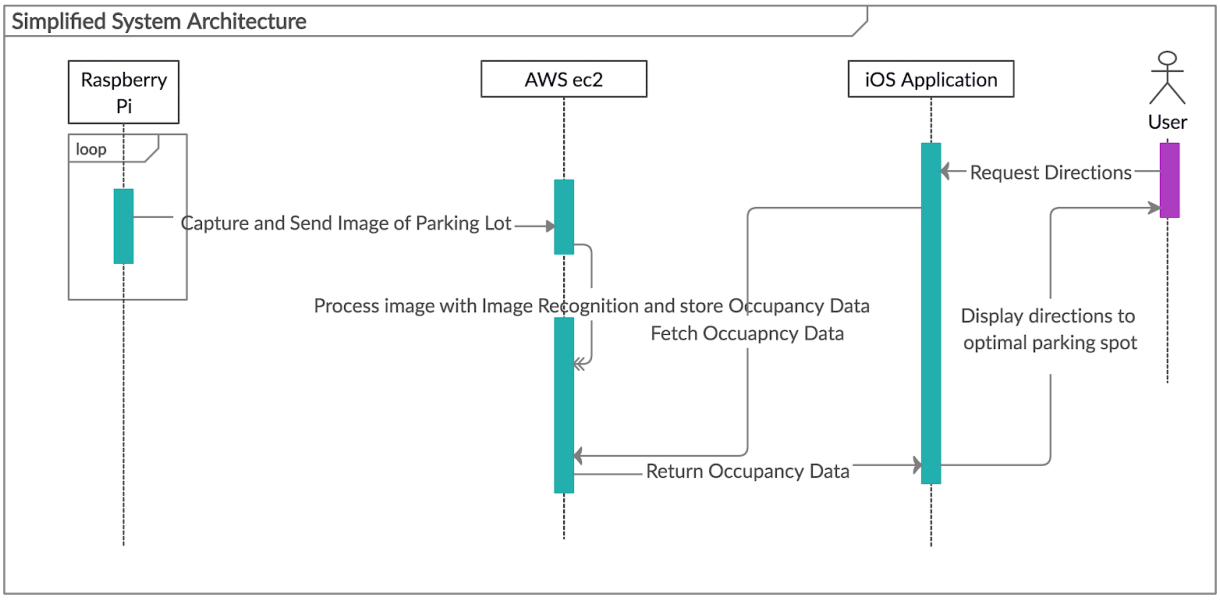
2. Methods / Results / Approach

2.1. Methods

2.1.a. System Architecture - An Overview

Our system has four core parts: Raspberry Pis, Amazon EC2 virtual machine, the weighted graphs representing parking lots, and the iOS application. Our system begins with the Raspberry Pis we install in parking lots to capture and send images of the parking lot to the Amazon EC2 instance. The next part of our system is the Amazon EC2 instance, which runs the ,where the images of parking lots are processed by an object detection script and converted to occupancy data. This EC2 instance also handles the storage of the occupancy data, and the communication with the Raspberry Pis and the iOS application.

In order to provide navigation in parking lots, we devised a method to represent parking lots as weighted graphs. These graphs provide the information necessary to calculate which parking spots are closest to the user, and to find the shortest path between two points in a parking lot. Lastly, the part of our system that the user interacts with is the iOS application, where the user can see real time occupancy data of parking lots and get directions to the optimal spot. The specifics of these system components and how they interact are defined in the sections below.



2.1.b System Component 1: Raspberry Pi

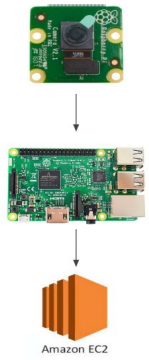
i. Component Overview

In order to capture images of a parking lot, we installed multiple Raspberry Pis with cameras attached in fixed positions overlooking the parking lot. The Raspberry Pis are responsible for capturing

and sending images to the server hosted on the Amazon EC2 instance, and providing information about the parking lot in which the images were taken.

ii. Tools

- ➢ Raspberry Pi4
- ➢ Raspberry Pi Camera (1080p)
- ➢ OpenCV
- ➢ VNC Viewer
- ➢ Python3



iii. Capturing Images

Our goal was to take pictures every few seconds in order to provide real time occupancy data for each parking lot. This meant that our system needed to continuously snap pictures and send them to the AWS EC2 instance. This was accomplished by looping our script to take a picture every "x" seconds. The frequency with which images are sent can be modified according to the availability of open parking spots in a lot. For example, during hours of light traffic in a parking lot the system can send a picture every 20 seconds, but during high traffic situations (ex: an event) the frequency can be reduced to 10 seconds. By varying this frequency based on demand for parking spots, we can prioritize the processing of parking lot images from lots with fewer available parking spots.

iv. Saving and Sending Images

The openCV library allows us to treat the video feed captured by the Raspberry Pi camera as an array. Each frame can be manipulated and saved as an image. So every so often, we can store the images onto the raspberry pi. Inside of our capturing images script, we have a function that will send the saved image to a web server hosted on AWS. Everytime an image is taken, it will be automatically sent to the server. Also, the images will be sent with a JSON file that will describe the parking lot name, camera number (we would have multiple cameras per parking lot), last update, and name of the image. An example of the JSON file sent with each image is provided below.

```
data = { 'pkl_name' : pkl_name,
         'camera_name' : camera_name,
         'last_update': 1,
         'image' : jpg_to_text }
```

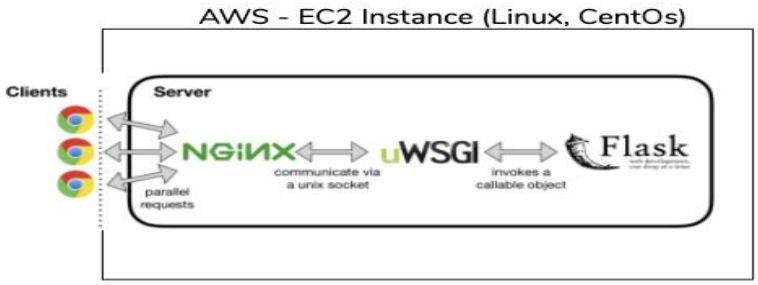
2.1.c System Component 2: Amazon EC2

i. Component Overview

This component of the system is responsible for receiving images from the raspberry pi, processing these images with an object detection script to generate occupancy data, and sending this information to the iOS application. The responsibilities of this component are detailed further in the sections below.

ii. Communication Protocol: REST API

The REST API was created in order for the Raspberry Pi to send images of parking lots, and for the IOS application to fetch the current occupancy data. The API was hosted on an Amazon Web Services EC2 Instance where we created our server for our application. We used Nginx as a reverse proxy, uWSGI, and Flask to host our scripts that would handle both camera and IOS application GET and POST requests. All client requests would first be sent to the Nginx HTTP web server, the server would forward those requests to the uWSGI application server via a TCP socket. The uWSGI application server on start up boots up the Python Flask application which handles the requests that were sent from the Nginx server.



iii. Scripts Running on Amazon EC2

GetImageFromServer.py

This script takes the posted dictionary that the camera sends to the server and extracts the sent image. First, the extracted image is then saved as a png file in a folder called pkl_images. Then, the parking lot json file corresponding to that image's parking lot is updated to let predictor.py know a camera has just sent a new image.

predictor.py

Since we need a script to make predictions using the trained CNN model (see section 2.1.c.iv) the predictor.py script was created. This script runs as a service on the Amazon EC2 instance and parses through all parking json files within the pkl_metadata folder. The script opens each JSON file and checks the last_update key value to see whether or not it was updated. If the value is a 0 then the script continues on. However, if the value is a 1 then the image for that parking lot must have recently been updated. The new image is then loaded and is split into images to represent the spots in the lot using openCV. These parking spot images are then resized to 224x224 and sent through CNN. With each spot picture a

prediction is made and the isOccupied key value corresponding to that spot in the parking lot json file is updated to either a 1 or 0 to match the prediction. A 1 meaning that the spot is occupied and 0 meaning that the spot is free. A representation of a parking lots json structure is shown below.


```
{"cameras":[{"camera_name":"1","image_path":"/var/www/html/myproject/myprojectenv/pkl_images/lot49_1.png","last_update":0,"spot_tags":[0,1,2,3,4,5,6,7]},
{"camera_name":"2","image_path":"/var/www/html/myproject/myprojectenv/pkl_images/lot49_2.png","last_update":0,"spot_tags":[8,9,10,11,12,13,14]},
{"camera_name":"3","image_path":"/var/www/html/myproject/myprojectenv/pkl_images/lot49_3.png","last_update":0,"spot_tags":[17,18,19,20,21,22,23,24]},
{"camera_name":"4","image_path":"/var/www/html/myproject/myprojectenv/pkl_images/lot49_4.png","last_update":0,"spot_tags":[25,26,27,28]},
{"camera_name":"5","image_path":"/var/www/html/myproject/myprojectenv/pkl_images/lot49_5.png","last_update":0,"spot_tags":[15,16]}],"metadata":[{"bound_yx":
[715,378,72,52],"isOccupied":0,"tag":"0"},{"bound_yx":[623,372,74,60],"isOccupied":1,"tag":"1"},{"bound_yx":[515,374,82,56],"isOccupied":0,"tag":"2"},{"bound_yx":
[438,377,85,54],"isOccupied":0,"tag":"3"},{"bound_yx":[367,378,96,54],"isOccupied":0,"tag":"4"},{"bound_yx":[279,380,95,57],"isOccupied":0,"tag":"5"},{"bound_yx":
[191,384,97,56],"isOccupied":0,"tag":"6"},{"bound_yx":[101,383,102,59],"isOccupied":0,"tag":"7"},{"bound_yx":[648,419,144,81],"isOccupied":0,"tag":"8"},{"bound_yx":
[556,417,132,80],"isOccupied":0,"tag":"9"},{"bound_yx":[467,416,118,81],"isOccupied":0,"tag":"10"},{"bound_yx":[380,416,108,80],"isOccupied":0,"tag":"11"},{"bound_yx":
[290,413,101,81],"isOccupied":0,"tag":"12"},{"bound_yx":[213,410,68,79],"isOccupied":0,"tag":"13"},{"bound_yx":[84,388,98,96],"isOccupied":1,"tag":"14"},{"bound_yx":
[417,388,134,119],"isOccupied":0,"tag":"15"},{"bound_yx":[303,391,139,116],"isOccupied":0,"tag":"16"},{"bound_yx":[673,415,69,62],"isOccupied":0,"tag":"17"},{"bound_yx":
[560,400,117,85],"isOccupied":1,"tag":"18"},{"bound_yx":[474,421,94,60],"isOccupied":0,"tag":"19"},{"bound_yx":[393,416,96,69],"isOccupied":0,"tag":"20"},{"bound_yx":
[304,414,92,69],"isOccupied":0,"tag":"21"},{"bound_yx":[219,413,81,65],"isOccupied":0,"tag":"22"},{"bound_yx":[146,410,73,64],"isOccupied":0,"tag":"23"},{"bound_yx":
[41,398,86,73],"isOccupied":0,"tag":"24"},{"bound_yx":[551,468,108,69],"isOccupied":0,"tag":"25"},{"bound_yx":[436,469,115,65],"isOccupied":1,"tag":"26"},{"bound_yx":
[328,468,116,65],"isOccupied":0,"tag":"27"},{"bound_yx":[216,470,116,59],"isOccupied":0,"tag":"28"}]}
```


send_pkl_data.py

This script was created to handle the GET requests sent from the IOS application. The IOS application would send a GET request asking for a certain parking lot json file. The scripts load up the correct parking lot json file corresponding to the name sent in the request. The information is then sent to the IOS application in the same json structure shown in the last picture. The application uses this information to populate a parking lot image whether a spot is taken or not based on the isOccupied key value within the json file.

iv. Image Processing

In order to figure out whether or not a parking space was free or occupied we decided to utilize a convolutional neural network (CNN). The network would be built in Python while using the Tensorflow deep learning framework.

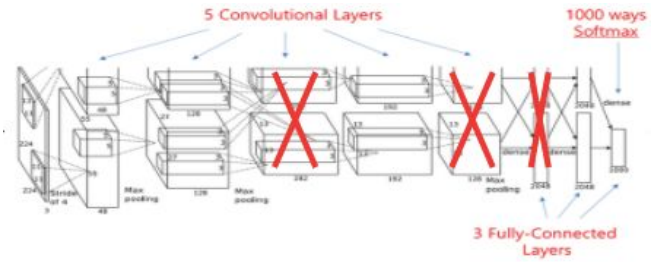
Dataset

The first step to before you can even build a CNN is to acquire a large enough and varied image set to train a model. Through doing some research online we came across the PKLot dataset which was created with the help of three Brazilian Universities, UFPR, PUCPR, and UEPG. The dataset contains close to 700,000 parking spot images that are from three different weather conditions (sunny, rainy, cloudy) and also included images with camera obstructions. [1]

Model

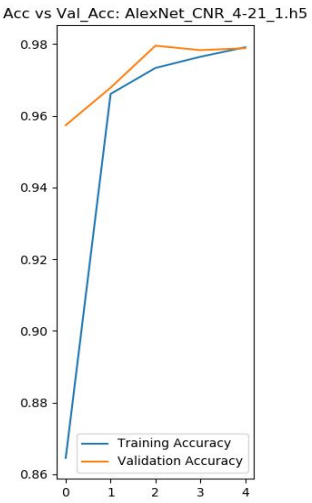
Since our project was only classifying whether a parking spot was free or occupied we decided to use the AlexNet CNN architecture for our network. This architecture is made up of 8 layers in total, 5 convolutional layers and 3 fully connected layers. In total it has a size of around 46 million weight parameters however we did not need the full size of this network. By removing 2 convolutional, 1 fully connected and reducing the density of both types of layers the model ended up with around 55,000 weight parameters. [2]

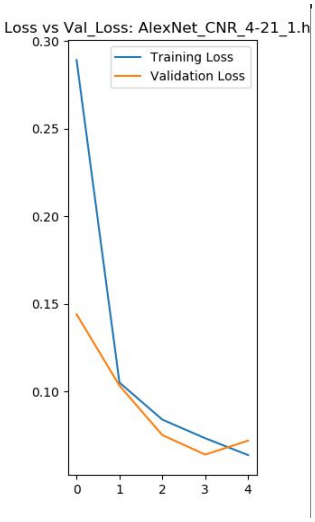
To shorten training time we used only 60,000 spot images to train our network and 50,000 images to evaluate the network. The model's accuracy ended up at around 98% and a loss of 0.037 when using the evaluation images. With the model at a point where it was working to show our proof of concept we decided to keep it and reinforce it in the near future if given more time.



(Part of the modified AlexNet Architecture shown above)

The graphs showcasing the training accuracy and loss vs its validation accuracy and loss are shown below. Keep in mind training accuracy and loss it generated based on images where the network knows the labels and validation accuracy and loss it based on images where the network does not know the labels to those images.





Essentially both graphs show that the predictions of the network whether on images with known labels or unknown have similar outcomes. While ending accuracy and loss values shown above do not mean the model will perform this well on all new data given, it allows the person training the model to see how the model does over its entire training time.

2.1.d System Component 3: Parking Lots as Weighted Graphs

i. Component Overview

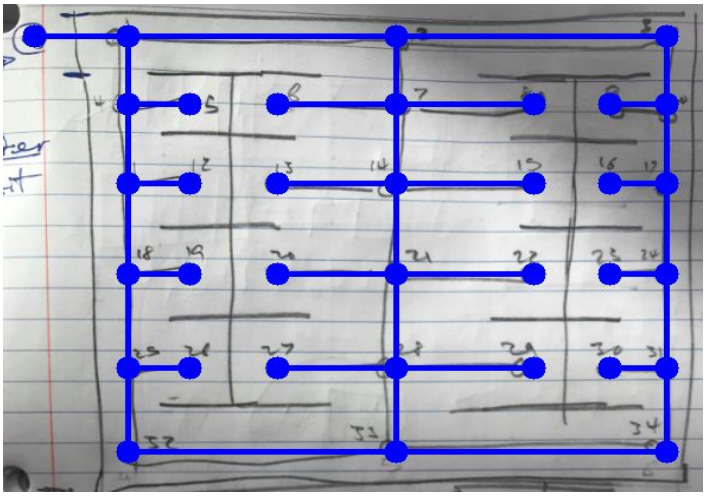
This component of the system involves representing parking lots as weighted graphs. These graphs are necessary to find the shortest path between two points in a lot, and provide the user with directions to the optimal parking spot. For each parking lot's weighted graph, there is a vertex within each parking spot and at every intersecting road within the lot. The weights of the edges between each vertex equal the distance between the vertices the edges connect.

ii. Creating the Graph

The first step in creating the weighted graph involved dropping points on a satellite image of a parking lot at every point there should be a vertex. In order to do this, we created a Matlab script that allowed us to plot points on an image, and record the x and y coordinates of all the points we plotted in a .txt file called *vertexData.txt*. The next step involves creating a .txt file containing the information about which vertices are connected by an edge and saving it to a file called *connectionsData.txt*.

Once the two aforementioned .txt files have been created, they are inputted into a script called *LotToGraph.swift* which processes the files and creates two matrices representing the weighted graph. For a parking lot with *n* vertices, each matrix will have dimensions of *n x n*. The first matrix is called the Weight Matrix, which stores the total distance in the shortest path from one vertex to another vertex. The second matrix is called the Path Matrix, which is used to find the vertices traversed in the shortest path from one vertex to another. In order to create these matrices, we used the Floyd-Warshall algorithm on the immediate connections and edge weights data. With these two matrices, we can now generate directions to and from anywhere in the parking lot.

A visual example of this programmatically generated weighted graph is shown below. As the image shows, there is a circle (representing a vertex) in every parking spot, beside each parking spot, and at every intersecting roadway in the lot. The lines connecting the vertices represent driveable routes in the parking lot.

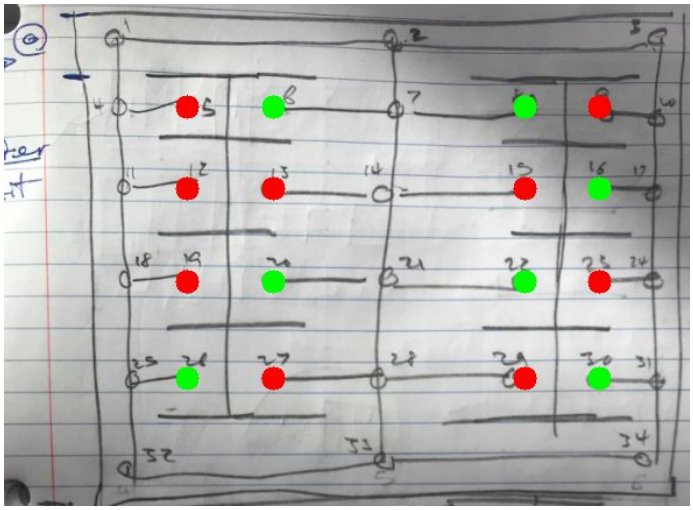


Programmatically generated weighted graph of a parking lot.

ii. Integrating Parking Spot Occupancy Data

When users request directions to a parking spot, the system needs to be able to sort the open parking spots by distance from the user's desired exit point. Therefore, this weighted graph also needed to store information about which vertices represent parking spots, and whether or not they are occupied. This was accomplished by manually creating a file called *spotToVertex.txt* which stores information about the direct mapping between parking spots and vertices.

In the iOS application, these weighted graphs are stored as data structures for each parking lot. Therefore, since the iOS application contains the real time occupancy data for each parking lot, it can input this data into the weighted graph and allow it to sort the open spots by distance from the user's desired exit point. The result of inputting this occupancy data into the graph and overlaying it over an image of our example lot is shown below.



Occupancy data overlaid over a parking lot image. Green and

red dots are open and occupied spots respectively

2.1.e System Component 4: iOS Application

i. Component Overview

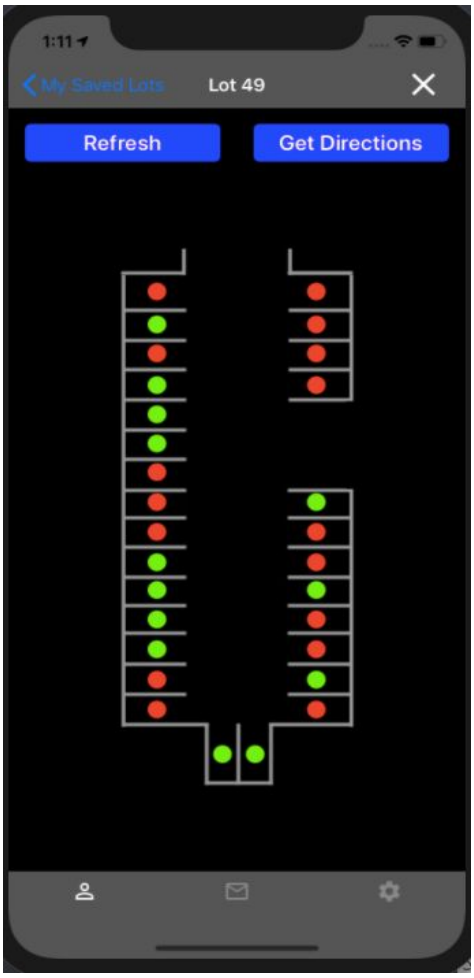
The iOS application, QuikPark, is the component of the system that the user interacts with. The application allows users to see real time data about which parking spots are occupied, and get directions to the parking spot closest to their desired exit point.

ii. Fetching Occupancy Data

In order to fetch the most recent occupancy data, the iOS application sends a GET request to the server hosted on the Amazon EC2 instance. Since our system provides data for multiple parking lots, this GET request contains information about the name of the parking lot for which it is requesting occupancy data. The response to this request contains a JSON file detailing which parking spots are occupied, and which are free. Then, this data is fed into the weighted graph data structure, allowing the system to sort the open parking spots by distance from the exits.

iii. Viewing Occupancy Data

The homepage of the application allows users to search for the name of the parking lot they want to enter (ex. Yellow Lot). Once a user clicks a certain parking lot, the application displays an image with parking spot occupancy data overlaid on an aerial view of the parking lot. In order to refresh this data, the user can press the "Refresh" button to fetch the most recent data from the Amazon EC2 instance. Once this data is received, the application renders a new image of the lot showing which parking spots are occupied. An example of this page in the application is shown below.

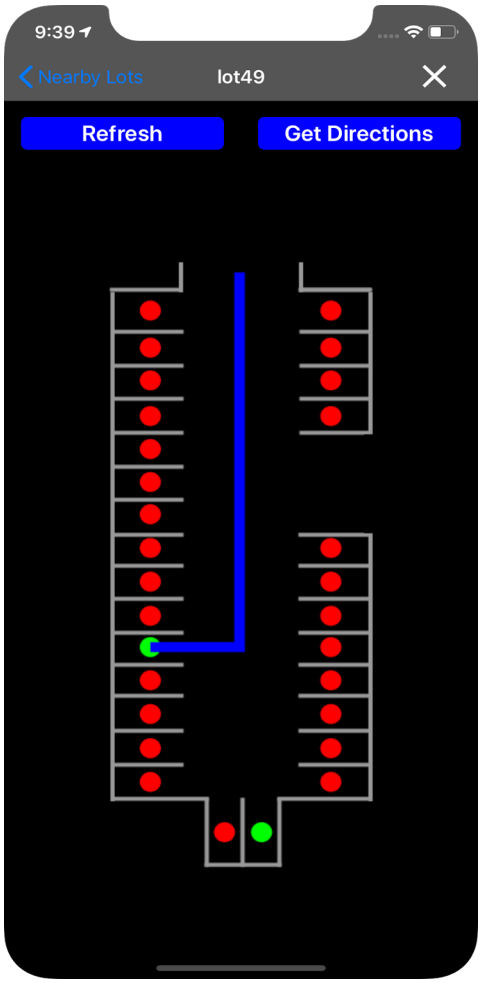


Parking spot occupancy data overlaid over Rutgers Lot 49

iv. Getting Directions to the Optimal Parking Spot

In order to get directions to the optimal parking spot, the user must first click his desired exit point after parking. For example, if a user wants to go to a building located by the top end of the parking lot shown above, he would first click this point on the application to set this as his desired exit point. After setting the desired exit point, the user can click the "Get Directions" button and see the shortest path to the optimal open parking spot.

The image below shows the result of clicking the “Get Directions” button after setting the desired exit point as the top of the parking lot. As the image shows, the application renders an image with a line drawn through the shortest path to the spot overlaid on an aerial image of the parking lot. Furthermore, the image shows that the application gives directions to the spot that is closest to the user’s desired exit point, the top of the parking lot.



Directions to optimal parking spot

2.2. Use of Standards

a. Standardized network technologies: Internet Protocol—IPv4 and IPv6, TCP

b. Standardized software development tools, and software environments: MATLAB, Python, Swift

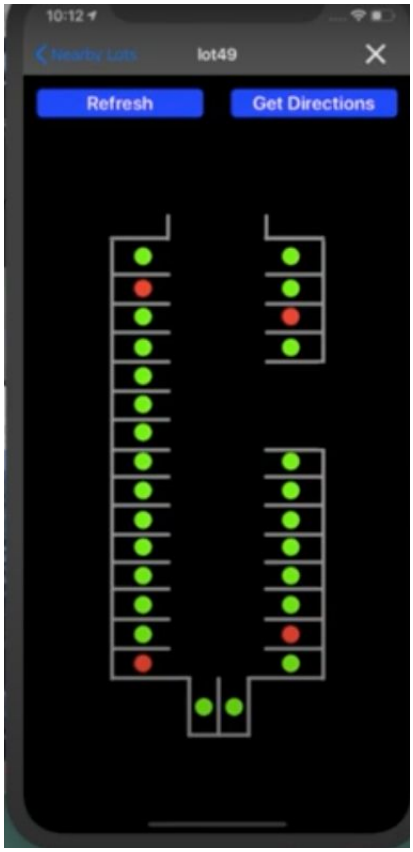
c. Hardware standards: Raspberry Pi V4 standards, Pi Camera

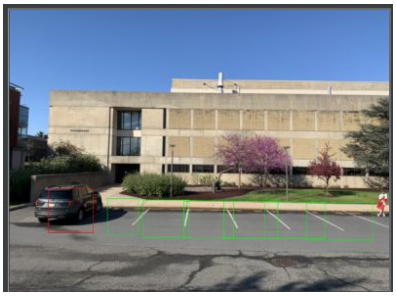
d. Open source standards, software, and operating systems: Linux (CentOs, Raspbian), Nginx server, OpenCV Image Software, uWSGI server, Flask, Tensorflow Deep Learning Framework

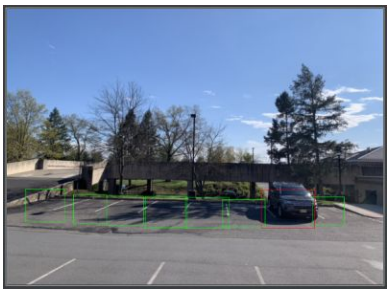
2.3. Experiment / Final Product Results

For our final results we tested our CNN network model on the images sent from our raspberry pi camera at Rutgers University, New Brunswick. We chose to use lot 49 on Busch campus due to its smaller size but before we could send images to our server for that lot we had to create a parking lot json file as described in 2.1.c. With the json file setup the raspberry pi camera took pictures every 10 seconds for half a day. Unfortunately, due to COVID-19 the lot activity was non-existent so we had to move around our own car to fill spots to gather the image data we needed. After gathering the first set of images to complete the parking lot setup we need to define the rectangle boundaries for each spot in the picture. Since the cameras are in a static position we would be able to reuse the rectangle boundaries for all new images sent from the camera. With the parking lot 49 completely set up on our server the camera could continuously update images and the predictor.py script on the server would generate the prediction for every parking spot in the pictures sent. The pictures below show the images taken with their rectangle boundaries while the rectangle colors represent whether or not the parking spot is free (green) or occupied (red). As you can see, we were able to achieve 100% accuracy with the few images taken. Along with images from the camera the corresponding application map generated from those images is shown. The colored rectangles in each camera image match up with the spot status circles in the application image.

Camera Image / IOS Application







3. Cost and Sustainability Analysis

A big component of this project was the economic feasibility in comparison to current alternatives on the market. Our main goal with using images in comparison to sensors was largely due to the long term expenses of maintaining such systems on a large scale. There are three main components when considering a cost and sustainability analysis which will be discussed below:

a. Economic Impact

i. For "Total Cost" it is important to note that our prototype only used one camera. In a small parking lot, we recommend the use of at least two cameras. The decision of additional cameras for the system is very dependent on the size and design of the parking lot. This would have to be analyzed by us before assessing a final estimate.

1. For example, in Lot 49 it would require at least two cameras due to the unique design. Each additional camera would cost an additional $97. This is because each camera needs its own Raspberry Pi, micro SD card, USB C power supply and a Pi camera.

- **Raspberry Pi V4 - $45**
- **Micro-SD card - $10**
- **USB C Power Supply - $15**
- **Micro HDMI to HDMI (only for development) - $10**
- **Pi Camera - $27**
- **Total Cost for System to Customer (1 camera): $97**
- **Total Cost for System and Development: $107**

ii. Comparison to the "TinyNode" smart parking solution by Paradox Engineering (A very popular existing smart parking solution). If we were to use Lot 49 to install this solution it would cost: Sensors ($130*30) + Network Infrastructures ($240) + Communications Gateway ($1099) = $5239 *[5]*

1. So, as you can see, the installation of QuikPark at Lot 49 would result in a cost of $194 (2 cameras). The cost of the "TinyNode" solution costs $5239 [4], so Rutgers would save $5045 if they used our system instead. Obviously the price would increase as the lot size increases, but our system would still be much less costly.

iii. Reduced expenses for maintenance, replacements, and installation. Our solution can work with minimal check-ups and operate for years without any maintenance.

b. Environmental Impact

i. Our solution has a minimal environmental footprint because we would be installing our cameras in obscure areas with a high field of view. The only resources used by our system would be an electrical power supply.
ii. Less power consumption due to needing to only power raspberry Pi's (5v power supply) opposed to many of sensors. We would use an existing power supply or a new power line instead of utilizing many sensors that would require a battery per sensor. Solutions who use batteries cause numerous environmental issues due to waste and incorrect disposal.
iii. Minimal waste as only a camera needs to be replaced and not multiple sensors.

c. Societal Impact
   i. Minimal disturbance to general population during installations
      1. In-ground sensors would disrupt traffic flow due to installation time
   ii. Addresses the need for efficient parking systems in parking lots
   iii. Reduces the overall time spent finding a parking spot thus reducing traffic in a parking lot
   iv. Reduces fear of parking violations due to built in lot availability based on time of day and parking permit

4. Conclusions / Summary

Despite the challenges we faced due to COVID-19, we were able to produce a high quality system that achieves our intended goal. However, this pandemic presented a few obstacles that limited the quality of our final project. Due to the lack of activity in parking lots, our group was unable to generate our own parking lot dataset of Rutgers. Had we been able to generate our own dataset, we could have used higher resolution photos than those from the PKLot dataset. In doing so we could have trained our model using the higher resolution photos, which would have resulted in a more accurate model.

Given more time we could have revisited the image recognition model and utilized a more modern architecture such as a residual network. While it is not certain that using a more modern architecture would increase our models accuracy it would definitely have given us more options to test so that we could find the optimal network model for our project.

The main motivation for this project was to improve upon existing smart parking solutions by creating a more cost effective solution. By using inexpensive Raspberry Pi cameras, image processing, and a free iOS application, we created a significantly less expensive smart parking solution by minimizing the hardware necessary to track and display parking spot occupancy data.

5. Acknowledgments

Professor Roy Yates - Advisor

Professor Hana Godrich - Course Coordinator

6. References

[1] PK-Lot Dataset Information, http://www.inf.ufpr.br/lesoliveira/download/pklot-readme.pdf

[2] CNR-Parking Research, http://cnrpark.it/

[3] OpenCV Tutorial, https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_gui/py_image_display/py_image_display.html

[4] Prior Solutions Prices, https://www.pdxeng.ch//wp-content/uploads/2018/05/Tinynode_A4-and-B4-product-brochure_2018.pdf

[5] TinyNode Parking Solutions, https://www.pdxeng.ch/tinynode/