Traffix

Ankita Chatterjee, Kaitlyn Liu, Zina Zarzycki

Department of Electrical and Computer Engineering,
Carnegie Mellon University

***Abstract*—Current fixed-interval traffic light implementations can be time-consuming and inefficient, making commuters wait for long periods of time in low-traffic situations. Traffix presents an optimized solution that uses camera footage from each side of an intersection to determine the number of cars and pedestrians on each side of an intersection and calculate the optimal light timing interval accordingly. This will reduce the average wait time of both cars and pedestrians at intersections, especially in sudden high traffic scenarios operating on fixed-interval light timings.**

***Index Terms*—Computer Vision, Machine Learning, Object Detection, Optimization, Printed Circuit Board, PyTorch, Q-learning, Reinforcement Learning, Traffic**

I. Introduction

The average American spends two-and-a-half work weeks in traffic per year. Congestion at intersections fluctuates frequently throughout the course of a day, and most traffic lights cannot anticipate sudden variations in traffic flow. This is especially evident in scenarios where roads are closed and detours are in place or other events that cause drastic changes in traffic density over a short period in time. This can include high density traffic scenarios such as road closures and detours as well as community events that cause an increase in traffic flow to a specific location.

Traffix aims to reduce traffic congestion by creating a smart traffic light that leverages machine learning and computer vision to identify cars and pedestrians at intersections and dynamically calculate light interval times using a Q-learning algorithm. This will benefit drivers by reducing wait times in traffic as well as benefit the city by providing a lasting system that increases city efficiency as the online machine learning model adapts to changes in traffic over time.

Although many cities have implemented traffic sensors that sense when no cars are present at a side of an intersection, then allowing the perpendicular traffic light to turn green, Traffix aims to provide optimization at another level by calculating future traffic through the use of API calls as well as an online machine learning model.

Even Los Angeles, one of the most congested cities in America, has made attempts to synchronize traffic lights using a system of induction sensors, however no attempts to use machine learning to optimize traffic have been made. The California Department of Transportation has indicated a need for AI to help reduce traffic, suggesting a need for a system like Traffix to optimize traffic using machine learning [1].

Overall, Traffix aims to leverage the power of machine learning to calculate and predict the most optimal light intervals to minimize wait times for commuters overall.

II. Use-Case Requirements

Because transportation networks can result in such a wide variety of dynamic real-world scenarios, it's important for us to carefully narrow the scope of our particular project and define some metrics that can help us evaluate the success of our final product. First of all, the exact physical layout that we are aiming to simulate is a four-pronged, two-way, two-lane intersection with traffic lights without protected left turns. To help visualize this, you can think about the Fifth and Craig intersection a few blocks from campus, as shown in Fig. 1. below. Additionally, we are only going to be considering day-time and high-visibility scenarios, because it will be too difficult to process low-light camera footage given this course's constraints.

Fig. 1. Google Maps satellite image of Fifth & Craig intersection with traffic overlay.

We have decided to focus our optimization efforts on situations in which there are at most 20 cars on each side of an intersection, with any scenarios above that being lumped together in our algorithms.

The most vital metric to aim for will be achieving a 10% reduction in average wait time as compared to a fixed-timing traffic light implementation, which we will measure using simulations. This is to ensure that commuters do in fact experience an efficiency improvement with our system. Lastly, we will need to ensure that our system adheres to all existing traffic laws to avoid creating unsafe scenarios for drivers and pedestrians.

III. Architecture and/or Principle of Operation

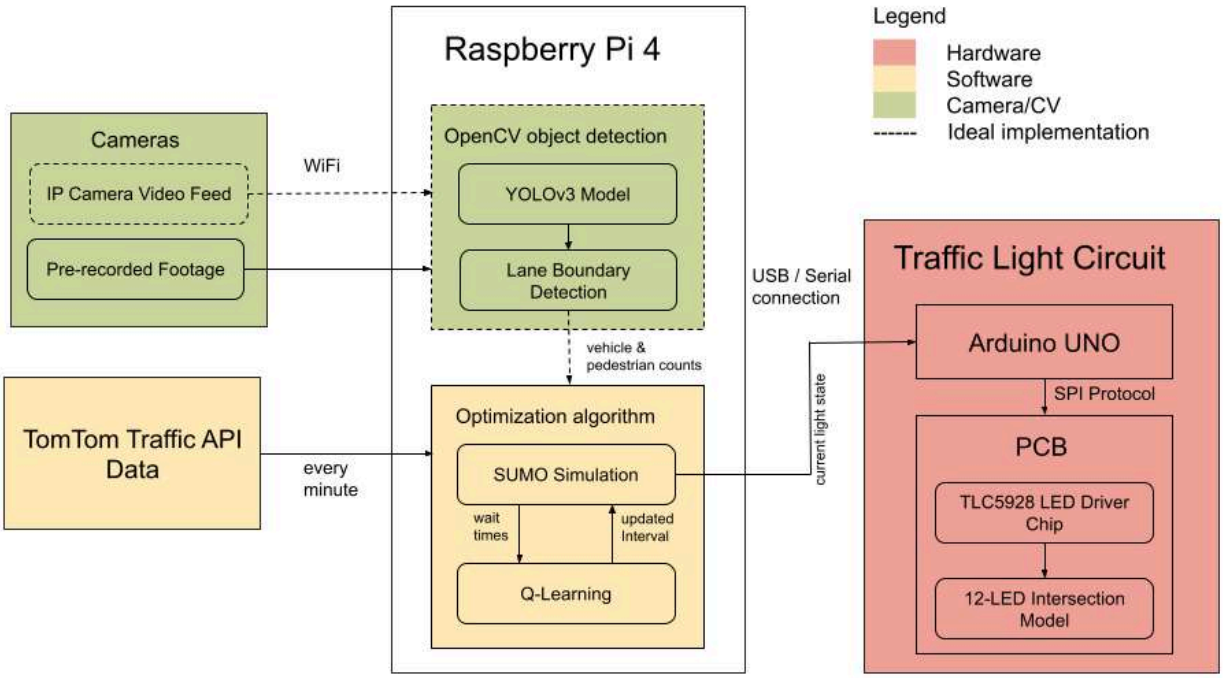


Fig. 2. A block diagram of the overall system.

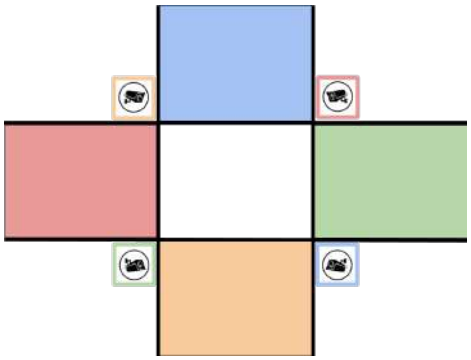


Fig. 3. Camera setup configurations. The color of each highlighted section corresponds to the camera that will be filming it. We went with a 4-camera setup.

The overall architecture of the system is described in Fig. 2 above. We did not end up using video feed from internet protocol (IP) cameras placed at each side of the intersection (as shown in Fig. 3), as we had difficulties accessing the RTSP (real-time streaming protocol) URL from the Raspberry Pi 4B. Instead, for demonstration purposes, we filmed videos on all 4 sides of the intersection at the same time, which was then processed using our object detection algorithm to obtain vehicle and pedestrian counts. Those counts would ideally be used along with traffic flow data from calls to the TomTom Traffic API by our optimization algorithm to determine whether or not the light should change state. The adjusted light state used by the SUMO simulation, which computes the wait times of each car at the intersection. Those values are fed back into the optimization algorithm. We also communicate the updated light state at each simulation step to the traffic light circuit, on which the corresponding LEDs light up accordingly.

There are three main parts to this system: the object detection model, the optimization algorithm running on the RPi, and the traffic light mockup circuit.

We ended up not fully integrating the object detection model with the optimization algorithm in our final demonstration for two reasons.

The first was that the latency of the object detection model when run on the RPi fell very far short of our latency requirements, to the point where it would compromise the accuracy of our simulation. The overhead added by processing each video frame would have added more than ten seconds of delay to the simulation, which is supposed to match real-time conditions as closely as possible.

The second reason is that the cars and pedestrians detected in the pre-recorded videos do not match the behavior expected by the simulation, as they move according to the actual light timings and not those calculated by our optimization algorithm. As a result, the wait times calculated by the simulation would not be accurate.

Instead, we included a user interface for the simulation that would prompt the user every minute to specify how many cars and pedestrians to simulate on each side of the intersection.

This served as a stand-in for the object counts determined by the detection model, and we demonstrated the detection model separately.

The object detection algorithm uses the YOLOv3 pre-trained classifier instead of a Haar cascade classifier to determine the number of vehicles and pedestrians waiting at each side of the intersection.

The optimization algorithm uses the vehicle and pedestrian counts from the object detection algorithm (for current traffic at the intersection) as well as traffic flow data obtained from API calls to the TomTom Traffic API (for future traffic at the intersection) to determine when the light should change. The SUMO traffic simulation platform is used to train the optimization model, since we used it to obtain the average wait time for cars and pedestrians. This metric is the reward function for the Q-learning model we employ for the optimization.

The traffic light mockup circuit consists of a custom-designed printed circuit board (PCB) mounted on an Arduino Uno. The RPi communicates with this circuit via a Serial connection to the Arduino, telling it when to change traffic light state. These light changes are reflected in the LED intersection on the PCB.

In our final product, the principles of engineering that we used the most were breaking down a large system into multiple different components that could be worked on and debugged separately, then integrated together once complete. We also implemented a Q-learning optimization model, experimenting with different action and state representations to determine what implementation would give us the best optimization.

Many of the principles of mathematics that we used came into play for our object detection model, where we used geometric concepts in 3-dimensional spaces to determine lane boundaries for each side of the intersection. Using those concepts, we were able to only include cars detected within our specified lane boundaries in our final counts.

The main applications of principles of science in our project had to do with testing and data collection. In accordance with the scientific method, we made sure to design our testing approaches in such a way that we were only manipulating one variable at a time. For example, when we tested our system's wait-time reduction, we held all variables constant in the simulation besides the swap between controlling the light timings with a fixed-time interval and our algorithm. Additionally, we made sure to collect different sets of data for the purposes of testing our object detection model.

IV. Design Requirements

We have compiled a list of various design requirements that we need to meet in order to satisfy the use case requirements we highlighted in section 2.

For our overall system, we want to keep the total delay (i.e. the time between initial frame capture and calculation of traffic light state change) below five seconds, as we intend to recalculate the light state every five seconds. We were conservative with this requirement as we do not yet know what kind of overhead we will see with both the object detection and optimization algorithms running on the Raspberry Pi. Five seconds is a period of time that is short enough for traffic conditions to remain mostly the same but long enough to account for any unanticipated overhead.

For our object detection algorithm, we want to be 90% accurate with the number of cars that we detect on each side of the intersection and 80% accurate with the number of pedestrians. This is in order to satisfy our requirement that the commuters that interact with our system feel that the light timings reflect actual traffic density conditions. Similarly, our object detection algorithm should be able to detect at least 10 cars on each side of the intersection to ensure that the system works in relatively high-traffic conditions.

Our optimization algorithm must demonstrate a 10% reduction in average wait time (for both cars and pedestrians) as compared to a simulated fixed-time light implementation using the same vehicle and pedestrian counts over multiple traffic cycles (2-5). This is to meet our requirement that there is in fact a noticeable improvement in wait time for commuters and pedestrians that use our system, and that the objective of our system — to make traffic light implementations more time-efficient — is satisfied.

We also want to ensure the accuracy of our traffic light mockup circuit. It must accurately reflect the optimization algorithm output (i.e. lights should change state when the algorithm determines they should), and the delay between the state change determination and the physical light change must be less than 0.5 s in order to ensure timely reflection of the current conditions.

Finally, we want to ensure that our system meets our safety requirement, which is that no two perpendicular lights are green at the same time. In other words, our optimization algorithm must guarantee that perpendicular lights do not overlap their green light intervals. We also want to make sure that we have a "minimum" time for each light to be green so that vehicles and pedestrians alike are given enough time to cross; this minimum time will depend on average vehicle velocity through the intersection and be determined through traffic simulations for the intersection we have chosen for our system, Fifth and Craig.

TABLE I. SPECIFICATION VS REQUIREMENTS

Specification	**Requirement**
Average wait time	> 10% reduction
Object detection accuracy	90% vehicles, 80% pedestrians
Object detection range	≥ 10 vehicles on each side
System latency	< 5 s
Traffic Light Mockup Accuracy	100%
Safety	0 overlapping green lights

V. DESIGN TRADE STUDIES

A. Raspberry Pi 4 Model B vs. NVIDIA Jetson Nano

Because a large part of our project used the OpenCV framework and we trained and ran various different machine learning models, we seriously considered using the NVIDIA Jetson Nano due to its more powerful hardware.

However, the Raspberry Pi has stronger community support and none of our group members have experience with the Jetson platform. Furthermore, because we'll be making API calls from the RPi and interfacing with the SUMO simulation platform, the flexibility provided by the Raspberry Pi was preferable for our uses.

In hindsight, it may have been worth investigating the Jetson option further, as the object detection model was too computationally complex for the Raspberry Pi 4B to handle.

B. Haar Cascade Classifiers vs. YOLOv3 Model vs. YOLOv4 Model

For our object detection models, we chose to use Haar cascade classifiers rather than the more complex YOLOv3 or YOLOv4 models. In general, the YOLO models boast a higher accuracy than Haar cascades, especially in complex scenes. They are also able to detect objects at various sizes, rotations, and light conditions, while Haar cascades tend to be limited to detecting objects at specific lighting and positions.

On the other hand, Haar classifiers are far less computationally intensive than the YOLO models, which was a big plus considering our limited compute power.

Ultimately, we went with the YOLOv3 model because the accuracy of the Haar classifier was nowhere near meeting our design requirements (the accuracy was less than 50% due to an abundance of false positives in each frame.) We tested the YOLOv4 model as well, but the accuracy matched that of the YOLOv3 model (~90%) and the overall latency was higher (~6 seconds to process one frame as compared to ~4 seconds.)

C. Q-learning vs. Model Predictive Control

For our optimization algorithm, we examined many different solutions but mostly focused on machine learning versus model predictive control (MPC). After looking into MPC more thoroughly, we realized that it would be a great way to model traffic and allow us a lot of control over the constraints of the system, however we would need an accurate traffic model, which we cannot obtain even with simulation softwares and API data. Additionally, machine learning is much more adaptable to change over time and does not require an accurate traffic model. Existing literature comparing the two algorithms indicates that Q-learning is more effective when we lack a precise mathematical model to mimic the system [2].

We ultimately chose deep Q-learning for our specific type of machine learning model because it can handle complex scenarios and is a form of unsupervised learning, which is necessary in our scenario since we would not need to label what traffic timings are the most optimal ourselves. Instead, Q-learning allows us to implement a reward function based off of the wait time generated by our simulation software.

D. SUMO vs Aimsun

We chose to use SUMO as a starting point for traffic simulation in our project because existing research used this software for machine learning and saw successful results [3]. SUMO is also a free software and is open source, so we can make modifications to the code if we need to as well. We also believe that the TraCI library that comes with this software makes it easy for us to integrate with our optimization algorithm and allows us to train our model easily. We can use the TraCI library to programmatically step through and control the simulation and use the simulation to generate the average wait time of cars in the intersection for our optimization algorithm's reward function.

We also looked into Aimsun, another urban planning software that has similar features to SUMO but is commercial and more user friendly than SUMO. We ultimately decided not to use Aimsun for the time being because the student version provides limited features, however if we feel that SUMO is not sufficient or lacks documentation, we may switch to Aimsun. Aimsun also has a scripting library called Aimsun Next which would act similarly to TraCI if we decided to use Aimsun over SUMO, however we found that Aimsun Next does not have the function calls well documented to demonstrate possible inputs unlike TraCI. The full version of Aimsun is also not available for free, so we ultimately decided to choose SUMO.

E. HERE API vs TomTom API vs Google Maps API

There are many traffic APIs available on the market that provide data on traffic flow and car density at specific

locations. We narrowed down our research to the HERE, TomTom, and Google Maps APIs. We chose these because they had large user bases and a lot of documentation, which we felt would be useful when trying to integrate them into our project, since we had no prior experience with any of these APIs in the past. We ultimately decided to choose the TomTom API because the Google Maps API does not provide traffic flow data and mostly focuses on routing and map image data, which is not useful to our project and HERE provides similar data to TomTom while updating less frequently (every 30 seconds in contrast to every 60 seconds). Both HERE and TomTom provide the current speed of cars passing through a coordinate area as well as the free flow speed of cars in an ideal environment. The TomTom traffic API will be used to sync our simulation to match real life flow and density, allowing us to accurately determine what state the traffic light should be in.

F. Standard vs. Addressable LEDs

Initially, we had planned to implement the Traffic Light Circuit (TLC) mockup using addressable LEDs to achieve the various representations of traffic light timings. However, after further investigation, it has become clear that addressable LEDs cause a number of problems. The biggest issue is that each addressable LED draws too high of a maximum current. The Arduino UNO we will be using to control the light transitions cannot supply more than 40mA per output pin, and each addressable LED can draw up to 60mA. Therefore, using the addressable LEDs without an external power source is untenable. Therefore, we will be using standard LEDs instead, which typically only draw up to a maximum of 30mA, which is within the safe range for the Arduino output pins.

VI. System Implementation

The system consists of 3 main subsystems — the object detection algorithm, the optimization algorithm, and the traffic light mockup circuit.

A. Object Detection Algorithm

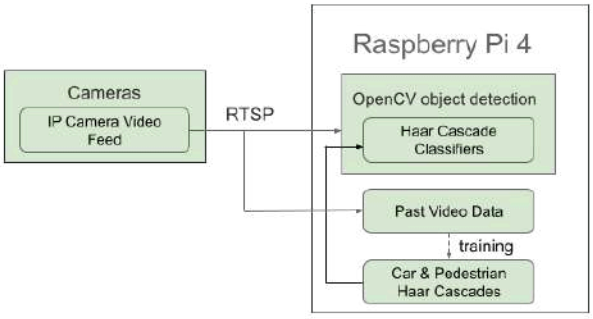


Fig. 4. Previous block diagram of object detection subsystem.

We modified the original approach to the object detection model, shown above in Figure 4, quite a bit. We discovered, after some discussion with the vendor, that the battery-powered IP cameras we purchased were not RTSP-compatible; as a result, we could not access the live video feed from the Raspberry Pi. We purchased alternatives, but while we were able to access the feeds from a personal computer, we were unable to replicate that behavior on the RPi. As a result, we decided to demonstrate our object detection model using pre-recorded videos from the intersection.

As specified previously, we used a YOLOv3 classifier for our object detection model. While we were initially planning on using a Haar cascade classifier that we trained ourselves, that classifier had very poor accuracy on videos we recorded at the Fifth and Craig intersection. We were unable to address the issues we faced with the pre-trained Haar cascade classifier, where there were a large number of false positives; we still encountered a large number of false positives on the model we trained ourselves. We may have seen better results if we trained the model on a larger dataset, as we only ended up using about 100 tagged images of the intersection. However, due to the high accuracy of the YOLOv3 classifier and ease with which we could use it in our project, we decided to change our model, especially because much of the support for training Haar cascade classifiers in OpenCV has been deprecated for several years.

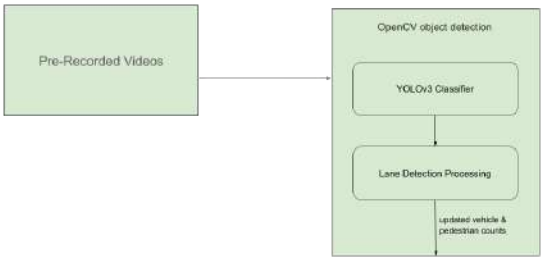


Fig. 5. Updated block diagram of object detection subsystem.

In our design report, we neglected to account for the fact that not all cars detected in a frame should be included in the overall vehicle count; only those detected in the lane that is waiting at the light should be considered.

To determine these lane boundaries, we initially attempted to use Canny edge detection, a well-documented CV algorithm that converts an image into a set of edges according to some threshold values, to find the edges of the lane and then transform those coordinates into lines using the method described in this article [4]. However, due to inconsistent lighting conditions and cases where cars would drive past the camera's field of view and block the lane, this approach did not provide us with reliable results. Instead, we ended up hard-coding the lines for the lane boundaries, which required a stable video as input without hand-shake. We felt that this

design choice was acceptable, as the ideal implementation of this product would anyways require cameras mounted in a stable location.

After determining equations for the lane boundary lines, of which there were typically two, we simply checked the following inequalities for each detected vehicle's bounding box coordinates, where m and b are the slopes and y-intercepts of each lane boundary line:

1) $y_{top\text{-}left} < m*x_{top\text{-}left} + b$
2) $y_{bottom\text{-}right} < m*x_{bottom\text{-}right} + b$

Fig. 6. An example frame from the Fifth and Craig intersection, with the lane boundaries marked in green. The cars included in the vehicle count are boxed in blue, while the cars not included in the vehicle count are boxed in red.

B. *Optimization Algorithm*

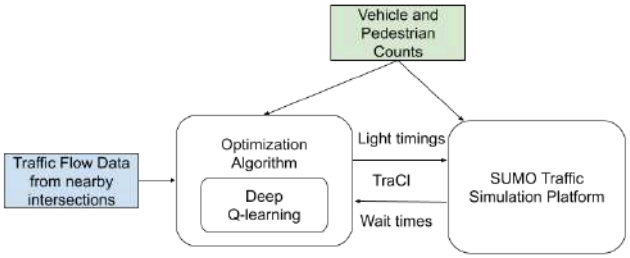


Fig. 7. Block diagram of optimization algorithm subsystem.

1) *Traffic API Integration*

We used the TomTom traffic API to gather data about the current state of incoming traffic to our targeted intersection. The API's *"traffic/services/flowsegmentdata"* endpoint provided data such as the current speed as well as free flow speed at the coordinate queried and a confidence level for the values returned by the call to the endpoint. See Appendix Fig. II for a more detailed JSON response from the API.

We used this data to accurately sync our SUMO simulation by syncing the flow speed with the calibrators in the simulation. These calibrators adjust how often cars spawn according to the flow (miles per hour of cars passing through the area) and cars per hour.

When the API outputs values with a confidence level over 0.5, we update the simulation to set the flow speed of the calibrator according to the flow speed from the API. We also take a baseline number of cars per hour that we estimated from regular traffic flow we gathered in our video footage and scale the baseline according to the ratio of currentSpeed output by the API to freeFlowSpeed. We currently have the program set up to allow the new vehicles per hour value set as a range of 0.8 to 1.2 times the baseline to prevent extreme fluctuations in car values.

2) *Q-Learning Algorithm*

We used a deep Q-learning algorithm with Pytorch and a single agent representing the light of the Fifth and Craig intersection we tested on. The Q-learning algorithm consists of a two layered neural network which has a number of input nodes to match the size of the state and a number of output nodes to match the number of actions the agent can take.

For the initial structure of the code, we referenced the classic Q-learning CartPole example [6]. We adapted this to work with the simulation and set a maximum duration for the episodes.

For our state data, we called functions to retrieve the state of the current SUMO simulation. The state data included the queue length for the north, east, south, and west sides of the intersection as separate parameters, the average speed of cars at each side of the intersection, the current phase of the light, as well as the time spent in the current phase of the cycle.

Our reward function is calculated by taking the mean wait time of cars at the simulated intersection detected by the lane detectors discussed in the *Training Simulation* section. We chose to only limit the area of the lane detectors at the intersection to 50 meters long to mimic the real life limitations of the object detection algorithm, since we would only have access to the counts of cars up to around 50 meters from the light via our camera feed.

$$L_\delta = \begin{cases} \frac{1}{2}(y-\hat{y})^2 & if\ |(y-\hat{y})| < \delta \\ \delta((y-\hat{y}) - \frac{1}{2}\delta) & otherwise \end{cases}$$

Eq. 1. Huber Loss Function

Our loss function was the Huber loss function, because we wanted to use a function that was less susceptible to outliers than Mean Squared Error because it treats values outside of a certain range linearly.

For our action states, we originally considered two different state implementations. We initially went with an action output of how long each parallel pair of directions (North-South, East-West) would turn green for. We considered that this would be a safer implementation since the light would have a fallback in the event that our optimization algorithm went down and could no longer output actions, since the light would continue changing phases according to the last input duration. However, after some testing, which we discuss more in section VII.B, we realized that the wait time reduction was minimal (less than 5%) with this implementation. We realized that the model was not reactive enough with this implementation.

Due to the minimal wait times, we decided to try a different action state representation to improve reactivity of the model. We instead sampled the optimization algorithm at a constant rate (2 seconds) and checked whether we should keep the current green light state or switch to a different state. One of the main considerations for this implementation was safety in the event that the optimization algorithm stopped functioning correctly, however we found that we can still have a fallback fixed time light with this implementation.

TABLE II. TRAFFIC LIGHT PHASES

Index	Description of Phase	Default Duration (s)
0	East-West green	100
1	East-West yellow	4
2	Buffer - all red	2
3	North-South green	100
4	North-South yellow	4
5	Buffer- all red	2
6	Pedestrian crossing	40

For example, if the traffic light was in state 0, the optimization algorithm would be sampled and if we received an action to stay East-West green, we do nothing and allow SUMO to continue cycling until the next iteration of updates. Otherwise, if we receive a state change, we set the phase of the light to 1, the East-West yellow phase. Since we are now in a transition state, we wait until we are no longer in a transition phase to call the optimization algorithm again. 6 seconds later, the simulation is in phase 3, which is not a transition state. At this point we start polling the optimization algorithm again to see whether we change state or not.

To prevent a permanent deadlock or light intervals that are too fast for cars to safely make it through, we set a minimum duration of 10 seconds and a maximum duration of 90 seconds. In the case that the optimization algorithm malfunctions, the fallback in this case is the fact that the light keeps a default duration of 100s green each way, so it would default to that if we do not receive actions from the optimization algorithm. We purposely set the default duration as an amount greater than the maximum duration so that the green phase could be anywhere between the minimum and maximum and we simply skip to the yellow phase to end the phase accordingly.

3) Training Simulation

We used the built in OSM tool that SUMO provides to generate a traffic network of the area near Fifth and Craig and modify it to include sensors. We took this network and simplified it by removing buildings and limited it to crucial streets in the area nearby. See Appendix Fig. III for a visualization of the OSM Web Wizard used to generate the area near CMU. The OSM software prefills most of the data for the simulation including right of way priorities, however we did have to make additions to the simulation.

SUMO can simulate both car and pedestrian behavior so we were able to spawn cars and pedestrians in. In an ideal scenario, we would have integrated the object detection model with the simulation by spawning in additional pedestrians and cars to sum to the total provided by the object detection model. We were unable to integrate the model, however we have the framework for spawning pedestrians and cars on all four sides of the intersection and randomized route generation for the cars to add variability to the simulation, bringing it closer to the real life environment.

To further increase real life accuracy, we added calibrators at points approaching all four sides of the intersection to simulate the flow of traffic received from our traffic API data, as mentioned previously. We sync with the API every 60 seconds. We also have lane detectors which we use to determine counts of cars as well as retrieve the correct cars in the simulation, mimicking behavior of the cameras in our system.

We then retrieve the average wait times and use them in our Q-learning algorithm as the reward function. See Appendix Fig. IV for a labeled diagram of the SUMO simulation. We then update our simulation according to the action output by the optimization algorithm and continuously step through the simulation to keep generating the output in sync with real life behavior.

4) Demo UI

As mentioned previously, we did not get output from the object detection algorithm into the simulation, so during the demonstration of our project, we took user input to replace the object detection output. We designed a UI in PyQt to get user input on how many pedestrians and cars to spawn in on each side of the intersection.

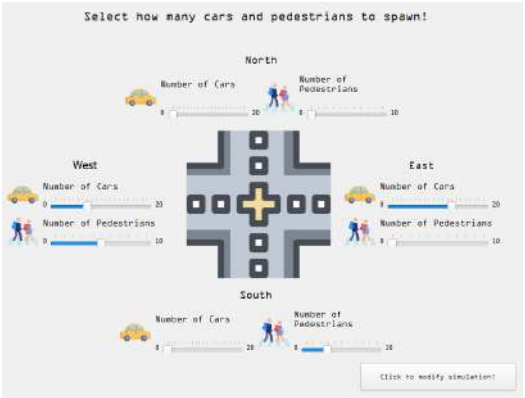


Fig. 8. The pop-up to input cars and pedestrians to spawn.

C. Traffic Light Circuit

The miniature model for the Traffic Light Circuit (TLC) can be broken down into three functional blocks: the Arduino UNO, the LED Driver, and the Traffic LED intersection. The broad view is that information about the current light state flows into this system from the RaspberryPi, which the Arduino then translates so that the TLC5928 Constant Current LED Driver chip can activate the Traffic LEDs in the optimized pattern. We will now discuss how the three blocks relate to each other in more detail. Refer to Appendix Fig. I to see how the overall circuit is wired together.

The Arduino and LED Driver pin connections can be visualized in Figures 9 and 10, respectively. We used four of the Arduino's digital GPIO pins to control the LED Driver, with the designated SPI pins handling the actual data transfer. The Arduino provides a reference clock for the Driver and uses its data transfer protocol to tell the Driver what lights should currently be active.

Additionally, the Arduino's 5V power pin will be connected to the Driver's V_{CC} input as well as the anodes of the LEDs. Resistor R_1, connected between the Driver's I_{REF} pin and ground, is used to set the constant forward current applied to each of its active outputs. We ended up using a 4.7kΩ resistor for R_1 to set I_{REF} to approximately 10.7 mA. The formula that was used to calculate this reference current is shown below:

$$I_{OUT(IDEAL)} = 42 \times \left[\frac{1.20}{R_{IREF}} \right]$$

Eq. 1. Ideal LED Driver output current.

Figure 11 shows how 12 of the Driver's 16 outputs will be wired to 12 LEDs, grouped into four sets of three to represent the Red-Yellow-Green light sequence for each side of a four-way intersection. The LEDs are connected to these outputs at their cathodes, since the Driver acts as a current sink rather than a current source [5]. This particular detail was something that we failed to account for in our initial circuit and corresponding PCB design process. Thus, the first set of PCBs we ordered did not work when the LEDs were wired in the way they had been designed to, but we were able to debug the problem by wiring up some breadboarded LEDs to one of the PCBs and seeing that these LEDs turned on as desired when their anodes and cathodes were connected in the way that the chip actually intended. We then modified the circuit design and the PCB layout to reflect this functionality. When the updated PCBs arrived, they ended up working exactly as we needed them to.

The schematics were created using Eeschema, which is the built-in circuit design tool for KiCAD, the software we used to do the layout for our custom-designed PCB that integrated the TLC system into a single unit.

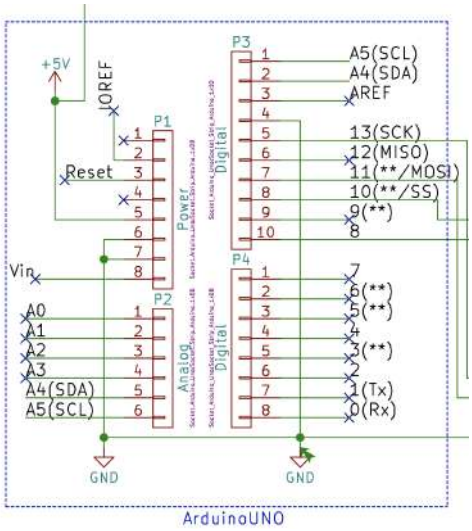


Fig. 9. Final schematic for Arduino UNO block.

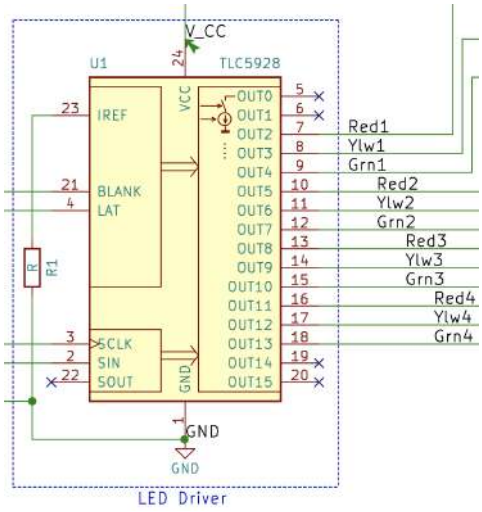


Fig. 10. Final schematic for LED Driver block.

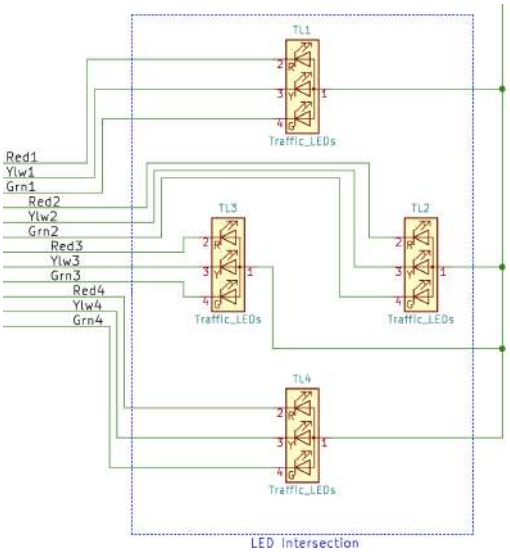


Fig. 11. Final schematic for Traffic LED intersection block.

VII. Test, Verification and Validation

In order to validate that our system meets our design requirements, we will be conducting a series of accuracy tests, latency tests, and integration tests across all four subsystems.

A. Accuracy

1) Object Detection Algorithm

As we described in section 4, our object detection algorithm needs to identify the number of vehicles and pedestrians on each side of the intersection with 90% and 80% accuracy, respectively. We tested this by looking at 200 frames captured at the intersection (50 from each side) and determining by eye the actual vehicle and pedestrian counts within the lane boundary. We then determine the average accuracy by comparing the vehicle and pedestrian counts determined by our model to the actual vehicle and pedestrian counts for each image.

This test is essential to ensure that our system determines light transitions based on accurate traffic condition information.

The table below shows how our object detection model accuracy varied over different iterations.

Table III. Specification vs Requirements

	Theoretical	Haar Cascade	YOLOv3
Car accuracy	90%	<50%	84%
Ped accuracy	80%	<50%	90%

We did not end up meeting the car accuracy requirements with the YOLOv3 model primarily due to cases where cars would drive in front of the camera's field of view and block the lane being analyzed. In those cases, we wrote code to maintain the previous vehicle count in order to avoid letting the count go to zero when there were in fact still cars waiting at the light. However, this meant that new cars could not be detected in that frame. We also faced some inaccuracies due to parked cars being detected within our lane boundaries, but were able to address this by adding additional boundaries to exclude parked cars.

We were able to meet the pedestrian accuracy requirements easily because no lane boundaries were required to determine the number of pedestrians; the video was already cropped to the pedestrian region of interest.

2) Optimization Algorithm

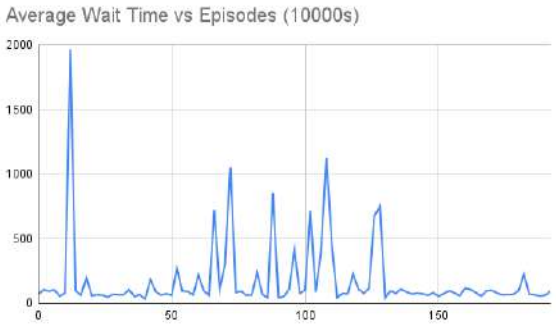
We aimed to improve wait times with our optimization algorithm by 10% according to our design requirements. We tested this by running our optimized simulation with our Q-learning model changing the state of the traffic light at Fifth and Craig against the same simulation without the model using a fixed time interval for the traffic light instead.

To verify the accuracy of the simulation's fixed time logic, we measured the duration of each phase for the real Fifth and Craig traffic light using recorded footage of the intersection. We noticed that the light does change according to the time of day and day of the week, however for simplicity when testing we decided to set our control as the most frequent interval pattern we saw in the recordings.

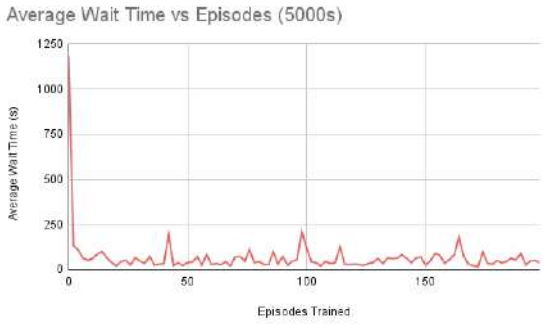
For our tests, we measured the average wait time of cars within the lane detectors over 30 periods of 3600 seconds, sampling every 60 seconds. During these tests, we randomly spawn various amounts of cars at each side of the intersection at random times to add variation to the simulation. We spawn these cars at the same probability in both the control test and experiment test. We then averaged the wait times of the 30 periods to produce the metrics and got an average wait time of 55.87 seconds for the control tests without the model and 49.10 seconds for the experimental tests using the model yielding a wait time reduction of 12.12%, which is above our goal of 10% for average wait time reduction. This relates to our use case requirement of the product resulting in a noticeable reduction in wait time for drivers.

During our initial tests for the Q-learning model using the light timing action representation, we thought we yielded a 48.90% wait time reduction, however after we began getting increasingly high wait time reductions, we realized that we had a bug in our system which constantly skipped the pedestrian state, making it an unreasonably short period of time that pedestrians would not feasibly be able to cross within. We fixed the bugs and only received a 5.4% wait time reduction, below our target of 10%. When testing, we noticed that the light timings output as the action for the Q-learning model converged to a constant value and only changed for extreme cases where there was a huge traffic jam in one direction that backed up for multiple roads. We realized this was likely due to the lack of responsivity in the model, since it would sometimes take a full cycle to see the effects of the action that the Q-learning model outputs, leading us to pivot to the alternative implementation that only outputs which sides of the intersection should be green, which is what resulted in the 12.12% wait time reduction.

During our testing for the new model, we also did some hyperparameter tuning and tested how different episode lengths affected the training.



(a)



(b)

Fig. 12. Average wait times over episodes trained for episode lengths of (a) 10000s and (b) 5000s.

We noticed that the 5000s episode length had much less variance, although both seemed to converge to their average wait time in about the same amount of time. We believe that this could potentially be caused by the periodic random exploration of the model sometimes leading to suboptimal behavior which propagates, and the longer the episodes occur, the more the traffic builds up, leading to the higher variance.

We also wanted to do more tuning on other hyperparameters such as the learning rate and gamma, however we ended up not having time to do so due to having to pivot our model towards the end and the new model being slower to train.

Our final model's main hyperparameters were a learning rate of 0.01, a gamma (future reward importance) of 0.99, and an episode length of 5000s.

3) Traffic Light Circuit

Testing the TLC was relatively straightforward. First, we ensured that the PCB was working correctly by running a testbench we wrote for the Arduino that attempted to activate each LED one at a time. Using this code, we discovered that our first circuit design was flawed, but got the LEDs to activate when hooked up differently than intended. We then used the same testbench on the new PCBs when they arrived, and found that they worked exactly as we had hoped. Lastly, we tested the interface between the TLC's Arduino and the RPi running our software sub-system and found that the serial communication was being received as intended.

B. Latency

1) Software

The software components of our system must have an overall latency of less than 5 seconds (i.e., the time between an image capture and the calculation of the light state change must be less than 5 seconds.) This means that it should take no longer than 5 seconds to both update the vehicle and pedestrian counts and calculate the new light state after a frame is captured.

On its own, the optimization algorithm had a latency of 102.4 milliseconds. We calculated this by logging the time stamps before and after the calculations necessary for one iteration of the optimization algorithm calculations for 20 different iterations. We then averaged the difference between the end and start times to get this metric.

The object detection model, when run on the Raspberry Pi, had a latency of 12 seconds on average to process four frames from four different sides of the intersection concurrently. It took approximately 4 seconds to process one frame alone. Clearly, this did not meet our latency requirements for the object detection model; we ended up demonstrating it on a laptop, instead, where the latency to process all 4 frames was ~2 seconds on average. This delay was calculated by starting a timer before the frame enters the processing function and then stopping the timer once the frame processing was complete.

2) Traffic Light Circuit

Since the data transfers from the RPi to the Arduino as well as from the Arduino to the LED driver chip both took mere microseconds to complete, we found that there was essentially zero latency added to the system by the TLC sub-system.

C. Safety

We ensured that our system was safe by ensuring that simultaneous green lights on perpendicular sides of the intersection were never possible. We did so by explicitly designing the traffic states to never overlap and also adding buffer states where all sides are red to prevent potential crashes. For our testing, we ran the simulation at a sped up rate of 1000 seconds per second using our optimization algorithm and ensuring we did not see warnings in SUMO, since we receive warnings whenever cars suddenly need to brake and there are dangerous conditions. This allowed us to modify our minimum duration for a single green light phase. We also observed to make sure that no states were being skipped over a period of 10 minutes of sped up simulation time. In the initial session of doing this test, we realized that there was a bug in the code where pedestrian crossings were being skipped, resulting in us realizing our initial wait time metrics were incorrect and modifying our implementation to actually meet our target metrics.

VIII. Project Management

A. Schedule

The Gantt chart in Appendix Table I shows the timeline for this project throughout the Spring 2024 semester.

For Kaitlyn's tasks, she ended up having to spend a lot more time training the model than anticipated and also took longer to create the simulation due to having to learn new software that does not have a lot of documentation. We also ended up not integrating the object detection model with the simulation code, so she created a UI to allow users to input values during the demo.

For Ankita's tasks, she ended up taking much longer than expected to code up the object detection model due to the change in implementation. Also, the first few weeks after spring break involved a lot of ordering and reordering of parts due to the issues we had with the IP cameras, which delayed finalizing the demo for the object detection model since we weren't sure if we were going to end up being able to use a live camera feed. She also had to retake the videos taken at the intersection multiple times, which required four people to film at the intersection at the same time and thus took some time to schedule. The demo ended up being finalized in the last two weeks of the semester after the final videos were taken.

For Zina's tasks, she ended up needing extra time to film the initial videos used to train and test the initial iterations of the object detection model. The circuit design process took longer than expected, but since she ended up using a surface-mounted chip for the implementation, there was no way to make a breadboarded model of the circuit to test things out before ordering and assembling the PCBs, thus this task was removed from the schedule. Having to re-design the circuit and PCB later on added time before final testing could take place, but everything still got done with plenty of time before our final demo.

B. Team Member Responsibilities

Our Gantt chart shows that our project is divided into five main categories: OpenCV detection algorithm, optimization algorithm, traffic simulation and API integration, Raspberry Pi integration, and traffic light circuit.

Ankita worked on the OpenCV detection algorithm as well as the Raspberry Pi integration and setup.

Kaitlyn worked on the optimization algorithm, traffic simulation, traffic API integration and demo UI.

Zina worked on the traffic light circuit and Raspberry Pi integration with Ankita.

C. Bill of Materials and Budget

The Bill of Materials is included in Appendix Table II. We ended up spending $323.08. We ended up not using the IP cameras. We also ended up having to purchase two sets of PCBs rather than the one set we had planned because the first circuit design was flawed.

D. Risk Management

1) Cameras and Live Video Feed

The biggest risk we had to manage was the fact that we did not end up using the IP cameras. Thankfully, we had the backup plan of using pre-recorded footage. We tried using indoor wired cameras instead, and powered them with a portable battery; however, we still were not able to access the live video feed from the Raspberry Pi despite being able to access it from a personal computer. Instead, we went out to the Fifth and Craig intersection with tripods and filmed all 4 sides of the intersection, then used those videos for our object detection algorithm.

2) Object Detection Algorithm

As described in our design trade studies in section 5, we decided to go with the YOLOv3 model over the simpler, less accurate Haar cascade. This is something that we provided as a backup plan in the design report, so it wasn't necessarily unexpected. However, it did mean that two or three weeks were spent in trying to tune the Haar cascade to meet our needs when we ultimately did not use it.

3) Optimization Algorithm

One of the main risks for this portion of the project was the algorithm not actually being more optimized than a fixed time system. Our initial implementation barely resulted in an improvement in wait times, however we did plan on testing out an alternative implementation if there was time throughout the semester and had allotted enough slack time to make the necessary changes. Since we realized the algorithm was not very optimized early enough, we were able to shift to the alternative implementation, which did meet our target wait time reduction and we still were able to do some of the hyperparameter optimization we initially planned.

Another risk related to the optimization algorithm was the simulation not working as we expected it to or not having the features we needed. We worked on the simulation earliest because we knew that exploring a new software would take time, so we were able to take time to ask users in the SUMO forum for help when we needed to. Even still, we did find out that there were unexpected behaviors in the simulation, such as calibrators clumping the spawning of cars towards the end of the time interval dictated instead of evenly distributing the spawning. Since we had researched so much into the software at that point, when a developer told us this on the forums, we were able to adapt the calibration of the software by modifying the intervals to be shorter and programmatically spawning new intervals using Traci instead.

4) *Traffic Light Circuit*

Although nothing went catastrophically wrong with the design and implementation of the TLC and we were able to have it working as intended by the end of the semester, there were still some challenges that required risk management. As mentioned previously, the initial PCB only worked when the LEDs were wired in a way that the PCB had not been designed for. We immediately re-designed and re-ordered these PCBs, but we had a backup in place for our demo, which consisted of the LED intersection being assembled on a breadboard with wires making the functionally correct connections to the PCB. Ultimately, we did not end up needing to use this breadboarded version, as the final PCB design had no issues, but it was helpful to have it on deck in case of emergency.

IX. Ethical Issues

There are a few potential edge cases for the operation of our product. We aim to improve the overall experience of commuters by avoiding situations where cars on one side of an intersection must wait unnecessarily long at lights where no cars are going on the adjacent side. At the same time, we want to avoid a situation where cars on a less busy side of the intersection are waiting for absurdly long periods of time in order to minimize the wait time of cars on the busier side, which is why we implemented a maximum wait time in our optimization algorithm.

In terms of public safety, our project has the potential to create unsafe situations where cars or pedestrians may not have enough time to cross the intersection or green/yellow lights on adjacent sides of the intersection are overlapping. These can cause accidents, which can be life-threatening. So in our optimization algorithm, we also have a minimum interval time (corresponding to the amount of time it takes to cross the intersection). This is somewhat of a tradeoff with the issue mentioned in the previous paragraph, because it may not always reduce the average wait time of those at the intersection.

We also guarantee that no two green lights are overlapping by having discrete light states as outputs of our optimization algorithm. There is no state where both sides of the intersection are green.

Another potential ethical concern with the ideal implementation of our project, involving the IP camera subsystem, would be security and privacy risks with stored IP camera footage. Malevolent actors may attempt to access the footage in order to determine the whereabouts of certain cars or people, so we would probably want to delete the stored footage after a certain period of time. At the same time, however, law enforcement may be able to use the footage as evidence to determine fault for car accidents and other such scenarios, so there are both benefits and drawbacks to this.

X. Related Work

While doing research on existing projects related to traffic optimization, we came across a research paper by Matt Stevens and Christopher Yeh which details using SUMO and Q-learning to optimize traffic [3]. This inspired some of our optimization algorithm's logic, however we believe that our project remains novel in its use of deep Q-learning over regular Q-learning and our integration of additional data through live traffic API data and image recognition. Additionally, we tested on a more precise simulation of real roads in contrast to their approach of a simple network of four intersections. Although aspects of our project are similar, we implemented everything from scratch since their research paper merely discussed the ideas and process they went through and does not provide direct sources of code.

XI. Summary

Traffix aims to improve the experience of the average commuter — drivers and pedestrians alike — by providing an optimized traffic light system that uses machine learning to determine optimal light timing intervals so as to minimize the average wait time of everyone at an intersection. Using cameras and traffic API data to determine the volume of traffic and piping that data into an optimization algorithm, Traffix offers a far more efficient alternative to conventional fixed-time interval traffic light implementations.

Stakeholders in our system, such as local transportation authorities that may be looking to improve traffic flow at city intersections, will appreciate our stated wait time reduction of at least 10%; also, our design requirements for minimum light time intervals and no overlapping green lights ensure that they will not be compromising on safety in pursuit of making the commuting process more efficient.

Ultimately, we were unable to fully integrate all the parts of our project due to limitations of the Raspberry Pi, however most of our other metrics achieved our target values or were close to the target values. We ultimately had a working optimization algorithm that could take in inputs from the object detection model as well as a working object detection model running separately. We also did have a working traffic light circuit that synced with the state from the traffic simulation.

A. *Future Work*

Some of the future expansions on this project we see includes expanding the system to multiple Q-learning agents rather than a single intersection. This would be possible by representing each light with its own agent and implementing a

pure cooperation system which would allow all the agents to receive the same rewards so that they don't try to compete against each other and optimize at the expense of other traffic lights. This is theoretically possible at the moment with our implementation, however we have not tested it on an actual simulation with multiple agents. The more agents in the network, the more we expect to see decreases in wait times.

Another potential addition to the system is speeding detection. We could potentially alter our light to adjust when we detect speeding vehicles, increasing road safety by preventing perpendicular cars from going too early and crashing into the speeding vehicle.

We could also use our object detection algorithm to detect emergency vehicles and prioritize their wait times over other wait times. This is similar to one of the initial ideas we had for a project at the start of the semester to alert emergency services if a car crash occurred at a red light.

As it stands, our project does not scale to lights with protected left turns or consider the fact that some cars waiting at one side of an intersection may plan to turn left or right rather than go straight as the simulation assumes. The object detection model could be modified to identify whether a vehicle's right or left blinker is on and communicate this information to the simulation.

B. Lessons Learned

Throughout the project, we learned many lessons about engineering as well as technical skills. One of the biggest lessons we learned is that we should always allocate more time for tasks than we expect. All three of us faced issues with tasks taking longer than expected. This was especially true with the simulation since none of us had experience using it and did not anticipate it taking that long to set up. For future students, researching technology they aren't familiar with is extremely important.

Another lesson we learned is to plan systems out as much as possible and do as much research as possible. When initially planning the optimization algorithm, we overlooked the fact that Q-learning requires discrete action states and wished to output a continuous integer instead. We had to adapt our model to be discrete for the first implementation of our algorithm. Additionally, we researched the cameras we planned on using to make sure they supported RTSP streaming and had accessible RTSP URLs, however we ended up finding out that this was not the case despite them being advertised otherwise. With some more research and planning, we could have potentially caught this earlier and found better cameras. We also did not anticipate the object detection model to be so computationally intensive, so reserving hardware earlier so we had access to better devices would have potentially resolved this issue.

Glossary of Acronyms

IP – Internet Protocol
OBD – On-Board Diagnostics
OSM – Open Street Map
PCB – Printed Circuit Board
RPi – Raspberry Pi
RTSP – Real Time Streaming Protocol
SUMO – Simulation of Urban Mobility
TLC – Traffic Light Circuit

References

[1] Queenie Wong, "California wants to reduce traffic. The Newsom administration thinks AI can help," Los Angeles Times, Jan. 08, 2024. https://www.latimes.com/california/story/2024-01-08/california-traffic-roads-safer-generative-ai-help

[2] L. Suryana, "Which one is better: Reinforcement Learning or Model Predictive Control? Inverted Pendulum — Case*," Analytics Vidhya, May 26, 2020. https://medium.com/analytics-vidhya/which-one-is-better-reinforcement-learning-or-model-predictive-control-inverted-pendulum-case-7fc29e52bbfb

[3] M. Stevens and C. Yeh, "Reinforcement Learning for Traffic Optimization." Accessed: Mar. 01, 2024. [Online]. Available: https://cs229.stanford.edu/proj2016spr/report/047.pdf

[4] Hardwick, Matt. "Simple Lane Detection with OpenCV." Medium, Medium, 22 Aug. 2018, medium.com/@mrhwick/simple-lane-detection-with-opencv-bfeb6ae54ec0

[5] Texas Instruments, "16-Channel, Constant-Current LED Driver with LED Open Detection datasheet (Rev. E)." https://www.ti.com/lit/ds/symlink/tlc5928.pdf?ts=1709206213832&ref_url=https%253A%252F%252Fwww.mouser.co.uk%252F

[6] A. Fakhry, "Using Q-Learning for OpenAI's CartPole-v1," Medium, Nov. 13, 2020. https://medium.com/swlh/using-q-learning-for-openais-cartpole-v1-4a216ef237df

Table I. Schedule

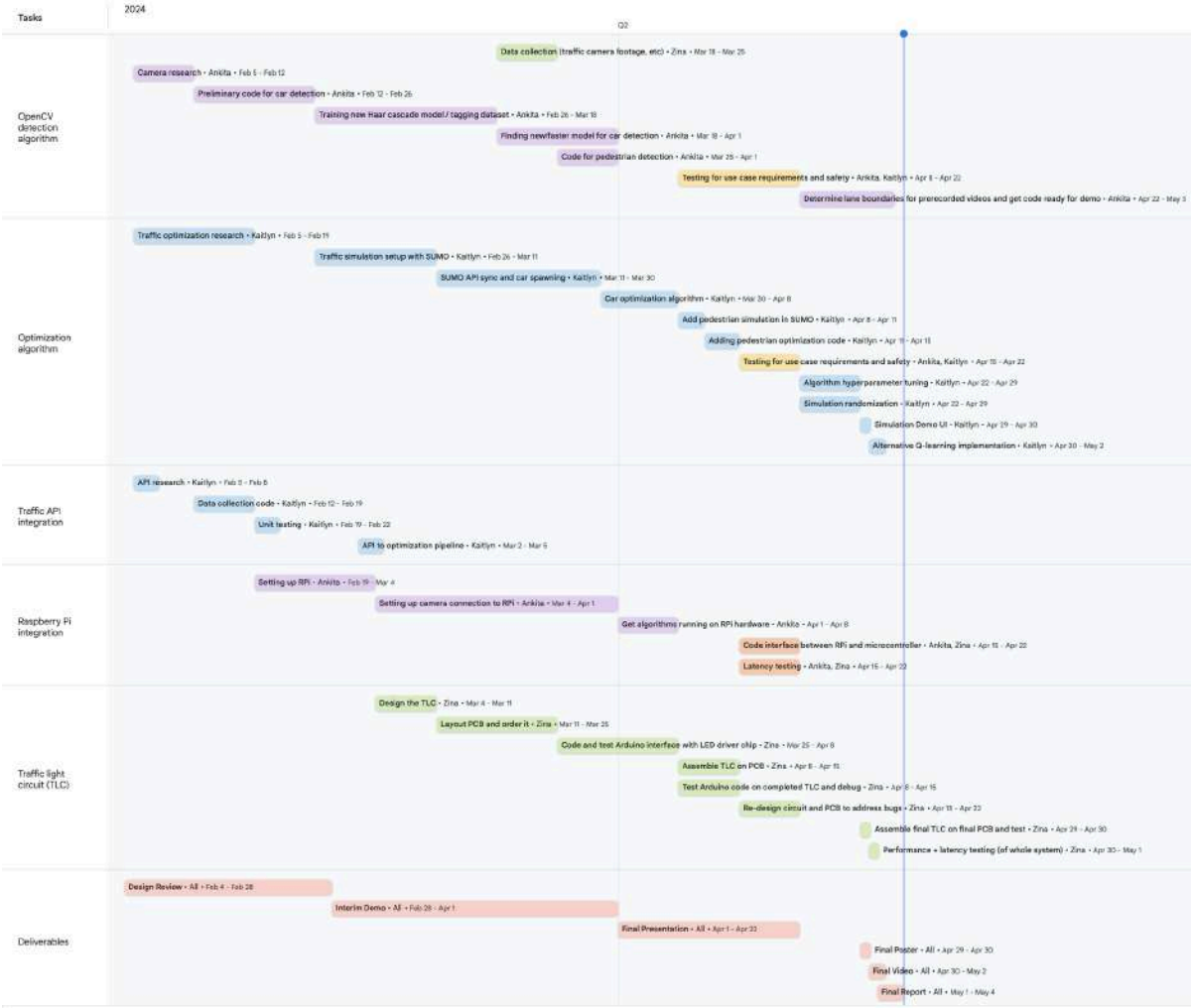


Table II. Bill of Materials

Subsystem	Item	Description	Model No.	Manufacturer	Quantity	Cost	Status	Used?
Camera	IP Camera	Wireless WiFi cameras for live video feed (for test purposes)	Argus 2E	Reolink	1	59.49	Received	Not Used
	IP Camera	Wireless WiFi cameras for live video feed	Q5	Mubview	1	37.09	Received	Not Used
	IP Camera	Wired WiFi camera for live video feed	2K	Mubview	1	18.54	Received	Not Used
	Pi Camera	Raspberry Pi camera (for testing purposes)		Raspberry Pi Fo	1	0	Received	Not Used
	BLE Camera Module	Bluetooth camera module (for test purposes)	ESP32-CAM	AITRIP	1	9.99	Received	Not Used
	Tripods	For stable filming at intersection		Victiv	3	52.35	Received	Used
Raspberry Pi	Raspberry Pi 4 Model B	Single board computer		Raspberry Pi Foundation	2	0	Received	Used
	MicroSD card	128 GB (for RPis)		Amazon	2	0	Received	Used
	MicroSD card	32 GB (for IP cameras)		SanDisk	2	13.69	Received	Not Used
Traffic Light PCB	Arduino UNO	Microcontroller board		Arduino	1	0	Received	Used
	TLC5928	16-LED Constant Current Driver	TLC5928DBQR	Texas Instrumen	4	1.58	Received	Used
	Arduino stacking header pins	Sets of pins to connect the PCB to the Arduino	1528-1074-ND	Adafruit	3	1.95	Received	Used
	PCB	First iteration of custom PCB for the TLC	2-Layer Prototype	OSH Park	3	58.3	Received	Not Used
	PCB	Second iteration of custom PCB for the TLC	2-Layer Prototype	OSH Park	3	70.1	Received	Used
	LEDs	LEDs for the mini TLC model (4 red, 4 yellow, 4 green)			12	0	Received	Used
	Resistor	4.7k-ohm resistor to set LED Driver current			1	0	Received	Used
					Total	323.08		

Figure I. Final Traffic Light Circuit Schematic

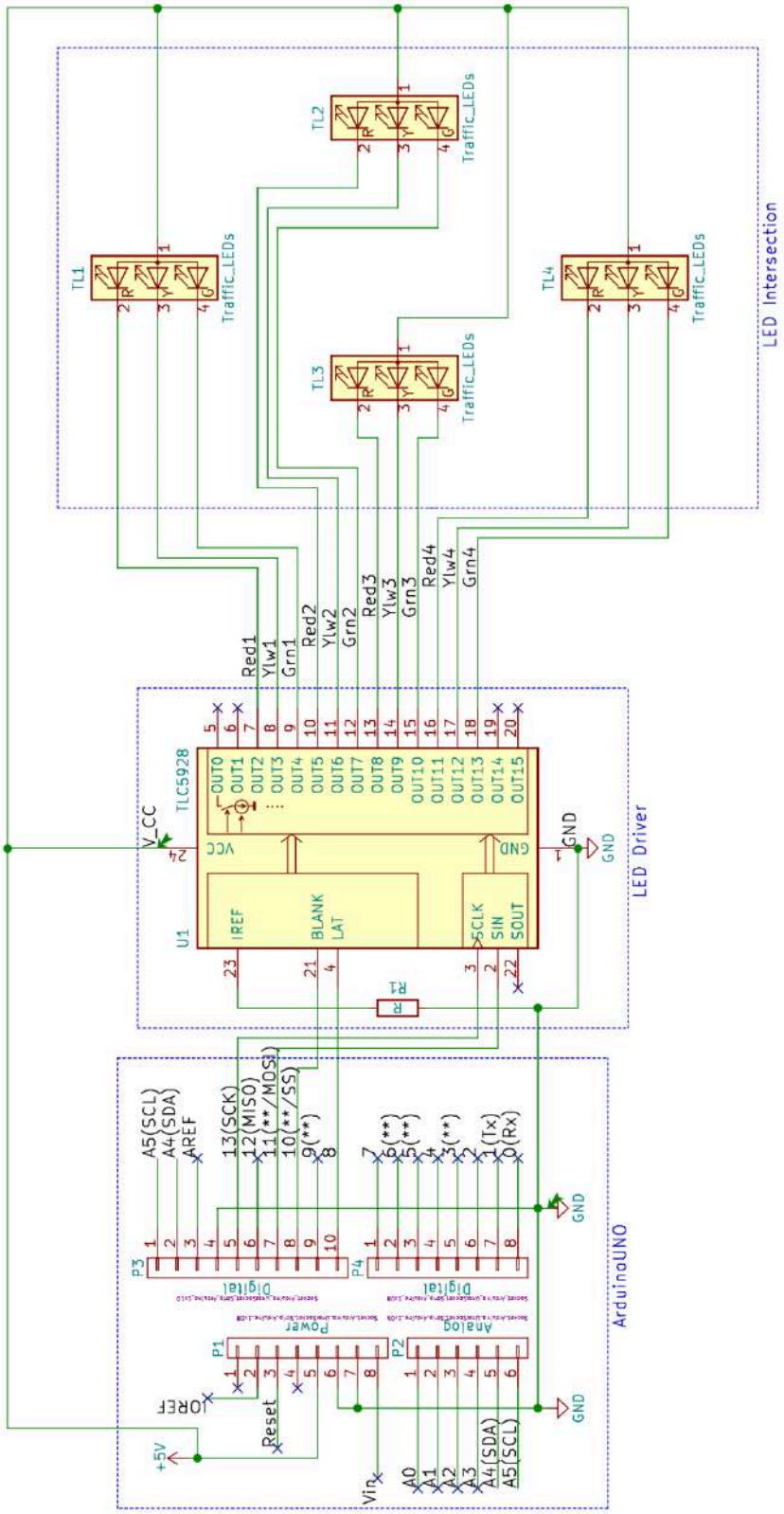


Figure II. TomTom API Response


```
"flowSegmentData": {
      "frc": "FRC3",
      "currentSpeed": 56,
      "freeFlowSpeed": 56,
      "currentTravelTime": 167,
      "freeFlowTravelTime": 167,
      "confidence": 1,
      "roadClosure": false,
  }
```


Figure III. OSM Web Wizard

Figure IV. SUMO Simulation Diagram