LEADER-FOLLOWER CAR PROJECT

April 24, 2019

Josh Baird
Evan Haskell
Justin Kur
Charlie Wingate

Contents

1 Project Vision

1.1 Background

As computers and algorithms become increasingly powerful, autonomous vehicles are becoming more of a reality. While these autonomous vehicles typically come heavily armed with tools so as to maximize their safety, computer vision is in itself a well-developed field, indicating that a great deal could be accomplished only using vision as an input to such a system. Our goal was to create a leader-follower car system, such that the leader car was manually controlled, while the follower car operates autonomously in order to follow the leader. We would prototype this system using Raspberry pi controlled miniature cars, and try to outfit the cars with the maximum degree of machine learning we could, to allow for a robust system which could be generalized.

1.2 Socio-economic Impact, Business Objectives, and Gap Analysis

Our project was research-oriented, so its business objectives are dependent to an extent on the effectiveness of the algorithms we explore. The project in a concrete sense allows for reducing the number of manual pilots of these vehicles. Since only one pilot needs to be the leader, and follower vehicles can be autonomous, the number of pilots is at least cut in half. Additionally, in theory using our same approach followers could be chained in convoy, allowing for a further reduction in pilots, and therefore expenses. In a broader sense, our general reinforcement learning approach and algorithms could be applied to a wider variety of robotic control systems, using only different reward signals and visual cues, and as such these future endeavors could benefit from seeing the failures and successes of this project.

1.3 Security and Ethical Concerns

The glaring ethical concern of this project is that follower cars learn their own control. To start, there is the concern of acquiring data in the first place. While some reinforcement learning algorithms are capable of learning from off-policy training data, many are only theoretically viable using on-policy data. Starting from only a randomly initialized model, the follower will only collect minimal training data before it has lost track of its leader, and training these episodes manually would be too taxing with our resources, so we were limited to off-policy algorithms. Then there is a concern that the off-policy training data does not generalize well enough to the follower's real environment. And of course, on a full scale system, one cannot afford to allow the repeated failures an untrained or poorly trained reinforcement learning agent would exhibit. The mitigation measures for this would mostly be domain dependent. In a true full-scale leader-follower vehicle system one would want fail-safes built into the follower to prevent it from taking dangerous exploratory actions. If our small-scale system is to function as simulated training data for a larger system, it would be imperative to use testing to determine that such training did indeed generalize.

As for security, the primary concern of our project is the communication between the Client Controller and the picars. In a true production system, such a system would need to be encrypted. As in our testing environment, such a system could use a dedicated network, so as to reduce the probability of a malicious attack on the system. Since any of the cars is capable of functioning as a leader or follower, losing access to even one picar would be considered a critical security failure.

1.4 Glossary of Key Terms

1. Pilot - The individual controlling the leader vehicle
2. Picar - A generic term for one of our testing vehicles
3. Server - The python application which runs on the Raspberry Pis
4. Client Controller - The C# GUI application which connects to the picar server and issues commands to the vehicles
5. Leader - The vehicle in leader mode, i.e. the vehicle being manually controlled
6. Follower - A vehicle in follower mode, i.e. one that tries to autonomously follow the leader.
7. On-policy Learning - Learning done with data collected by following one's own policy
8. Off-policy Learning - Learning done from experience that was not necessarily acquired by following one's own policy

2 Project Execution

2.1 Team Information

The Winter 2019 Leader-Follower Car Project is comprised of the following Team members:

Josh Baird	Backend and RL Developer
Evan Haskell	Backend
Justin Kur	Team Lead and RL Developer
Charlie Wingate	UX/UI Developer

2.2 Tools and Technology

Our project makes use of two Sunfounder PiCar-V vehicles, which we inherited from the previous semester's work. The picar vehicles are controlled using python 3 code–this includes Sunfounder libraries for low-level vehicle operations, gRPC protocol communication code, and our code which controls the leader and followers. The Client Controller is programmed in C# and features a GUI which allows for convenient connection to the picars, and of course it too uses gRPC libraries for communication with the picars. This general architecture was provided by the previous team, but we expanded it in many ways including porting the python code from 2.7 to 3.5, expanding the follower model to allow for reinforcement learning agents, and making enhancements to the GUI to test our new capabilities.

During our earlier tests, we experimented with Convolutional Neural Networks (CNNs), which were provided by the PyTorch library. These ultimately turned out to be too resource-intensive and finicky on our picar hardware, and as such the idea for using CNNs for supervised control or reinforcement learning function approximation had to be abandoned. Our final algorithms were mostly implemented by hand following pseudocode from Reinforcement Learning: An Introduction. [1]

2.3 Best Standards and Practices

We followed an agile development process for this project, which was important to accommodate our changing requirements and plans considering the experimental nature of our research. Our Sprints were divided into periods of 2 weeks, excepting the first two Sprints which were only 1 week apiece so as to evaluate the early deliverables more quickly. Due to the limited nature of the picar themselves, it was also important to consider upon each of our meetings who would be taking the cars. We usually made this call based off of whose work would be most closely tied with the hardware itself: usually this would fall to whoever was collecting training data or evaluating the performance of the reinforcement learning following algorithms.

3 System Requirement Analysis

3.1 Functional Requirements

1. The system shall support at least two cars: one to lead and one to follow.
2. The system shall allow any one registered vehicle to function as the lead vehicle.
3. The system shall allow any registered vehicle to function as a following vehicle.
4. The system shall allow the user to transmit directional movement instructions to the vehicle in leader mode.
5. A following vehicle shall avoid collisions with the leader vehicle.
6. A following vehicle shall maintain a safe distance from the leading vehicle.
7. A following vehicle shall stay within a reasonable configurable distance from the leading vehicle.
8. A following vehicle shall avoid collisions with foreign obstacles.
9. A following vehicle that loses visual contact with a leader should attempt to reestablish following protocol.
10. The system shall allow the user to enable logging and saving image data received on the remote client from the following vehicle.
11. The system shall allow a mirroring mode, wherein the following vehicle mirrors the instructions given to the leading vehicle on a delay.
12. The server application shall be able to log all movement inputs from a driving session.
13. The leader car shall be able to read and execute input data from a saved file.

14. The system shall be able switch leader and follower vehicles during program execution.
15. The system shall allow one user to control the leader vehicle at a time.

3.2 Non-Functional Requirements

1. The system shall be secure to external threats.
2. The application shall have zero severity level one defects.
3. The application shall support a remote control range of 20 meters.
4. The application shall have a mean time between failures of 30 days.
5. User input will have a response time of at most 250 milliseconds.

4 Functional Requirements Specification

4.1 Stakeholders

The primary stakeholders in the system are the owners and operators of the picars. While the previous iteration of this project was sponsored and guided by TARDEC, our work was more research and algorithm focused, meaning we could have greater influence on the wider field of autonomous robot control.

4.2 Actors and Goals

The main actor in the system is the pilot. However, both the leader and follower cars are considered agents, and thus actors. The follower is able to make its own decisions about how to act and thus influences the system. The leader also a passive actor in the sense that it provides data to the follower in the form of its tag location. The goal is for the following vehicle to maintain a safe distance and not crash.

4.3 User Stories, Scenarios, and Use Cases

Many user stories, scenarios, and use cases were already covered by the efforts of the previous team, especially in terms of basic GUI functionality. Therefore, we have ommited some of these in favor of focusing on the main ideas behind the follower behavior.

4.3.1 User Stories

1. As a vehicle pilot, I need to be able to control the directional input of a designated leader car.

2. As a following vehicle, I need to quickly and accurately determine the direction and speed the leader is navigating and move in the appropriate direction.

3. As a vehicle pilot, I want to be able to switch which vehicle is the leader on the fly.

4. As a vehicle operator, I want to be able to send precreated input data to the lead car.

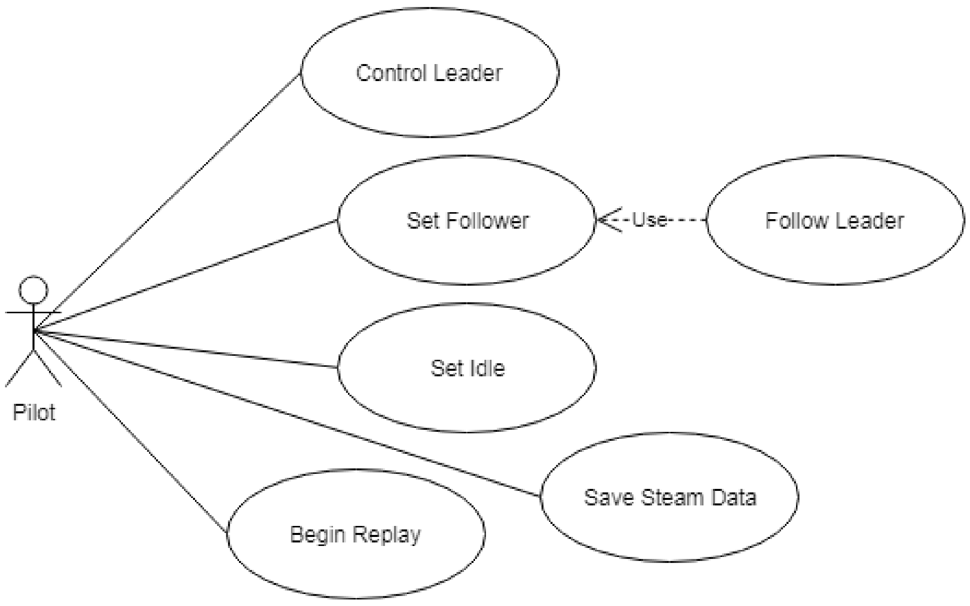
4.3.2 Scenarios

1. Jane owns a farm and wants to deliver a large amount of crops from one side of the farm to the other. Jane can pilot the lead vehicle herself, but she would be required to make many trips in order to completely ferry her goods from one side to the other. So instead, she can put her other vehicles into follower mode, and pilot the main vehicle herself, and lead the follower vehicles across in a single trip.

2. Robert wants to train an autonomous robot to follow a preset path. He can use the leader-follower system: by piloting the lead vehicle and setting another vehicle in follower mode, he can collect training data for the path-following robot, without requiring direct trial-and-error.

3. Steve works on more complex autonomous car systems, but during testing some of his sensors fail. His system can then fall back to this leader-follower approach, so that the damaged car can follow using only minimal visual cues, and still perform its mission.

4.3.3 Use Cases

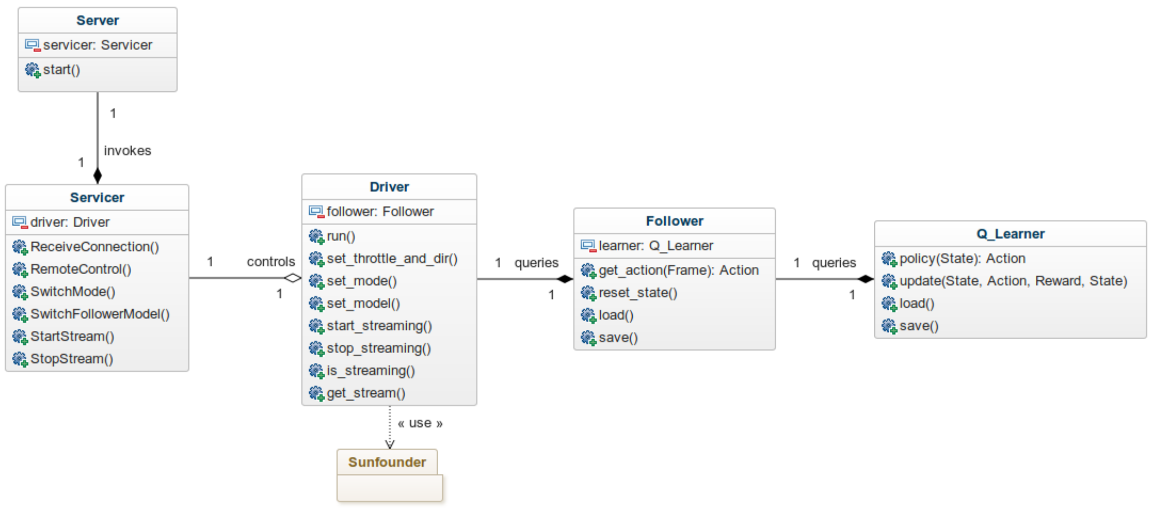


5 User Interface Specifications

Since our project was a continuation of previous semester work, we started with a functioning GUI which was largely sufficient for our needs. As such, we decided not to create specification for the existing framework.

6 Static Design

6.1 Class Model



6.2 System Operation Contracts

The high level contract of our system's operation is that the follower vehicle should maintain safe distance with the leader vehicle, and the vehicles should not collide with one another.

6.3 Mathematical Model

Considering our project is primarily trying to explore machine learning algorithms to solve this robotic control problem, the domain maps nicely to a mathematical model. When the picar is in leader mode, we consider its inputs to be commands from the Client Controller, a 2-dimensional vector $[throttle, direction]^T, throttle, direction \in [-1,1]$. Since a picar in leader mode is meant only to execute the commands it is given, the function f which gives this output is simply $f(x) = Ix$. Alternatively, when a picar is in follower mode its input (its state) is the features of the tag, as well as some state we retain to accommodate potential one-frame loss of sight from camera shake. The follower's Q-Learning table is itself a function: $Q(state, action) = value$. Thus to find the best action we return the action a which maximizes $Q(state, a)$ for a given state, which is found by iterating across all possible actions.

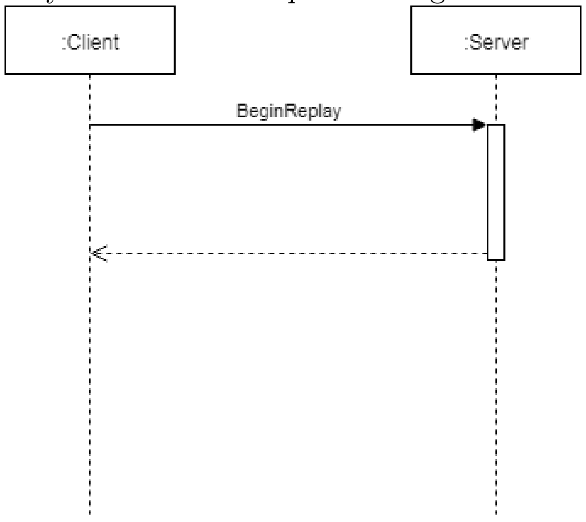
6.4 Entity Relation

This nature of this project is such that we do not really have different "entities" which relate to each other, with possible exception of the structure of the code itself. In that case, refer to section 6.1.

7 Dynamic Design

7.1 System Sequence Diagrams

Many basic GUI functions were already accomplished by the previous team, so we only include the sequence diagram for beginning a replay, a new feature we added.

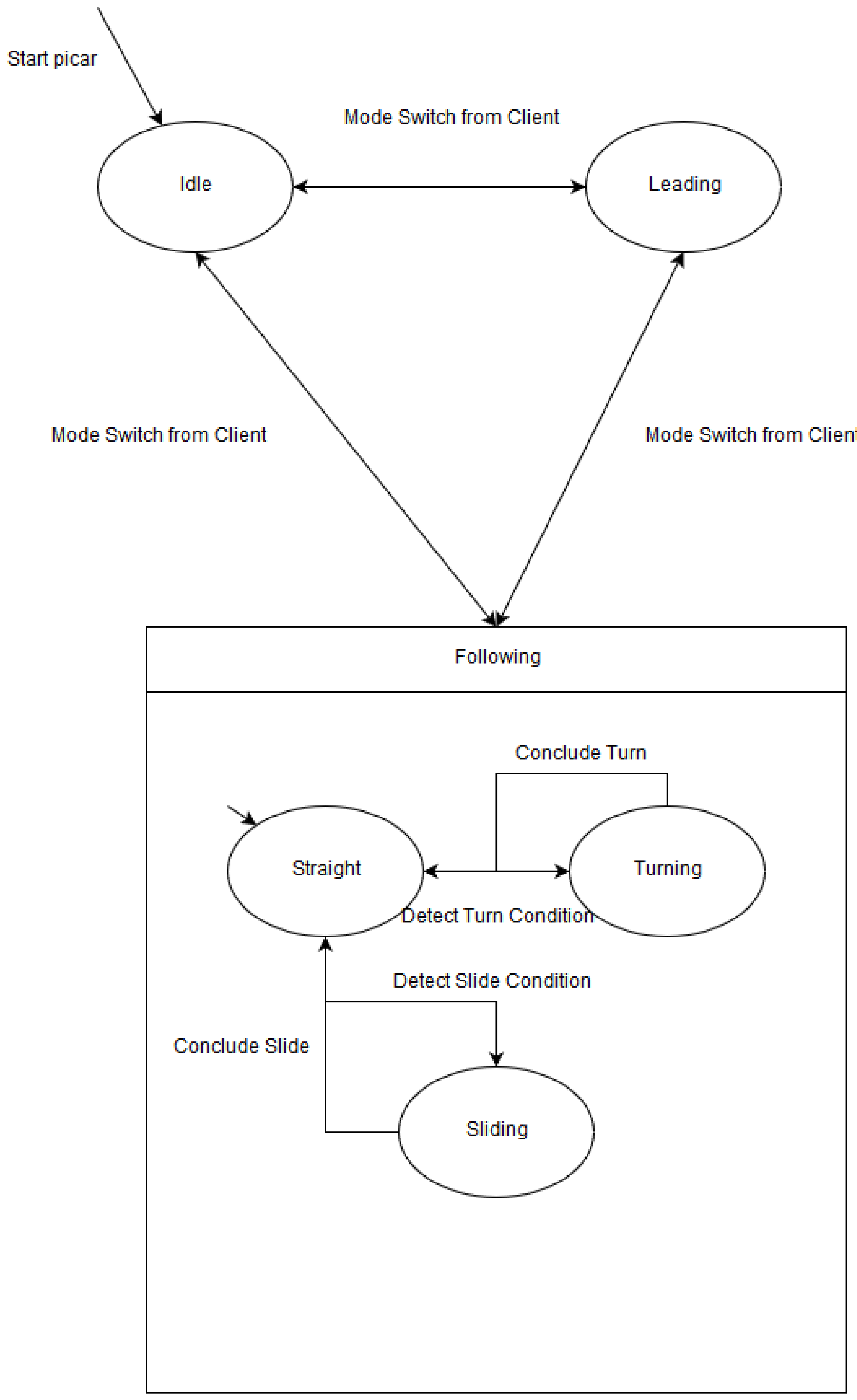


7.2 User Interface Specification

There are some basic behaviors that the UI must be able to perform. As we inherited a working GUI, we did not need to write the specification for many points, but we did add some behavior as specified below:

1. The user interface shall allow a series of previous inputs to be resent.
2. The user interface shall allow incoming picture and action data from a selected car to be saved to disk.

7.3 State Diagram

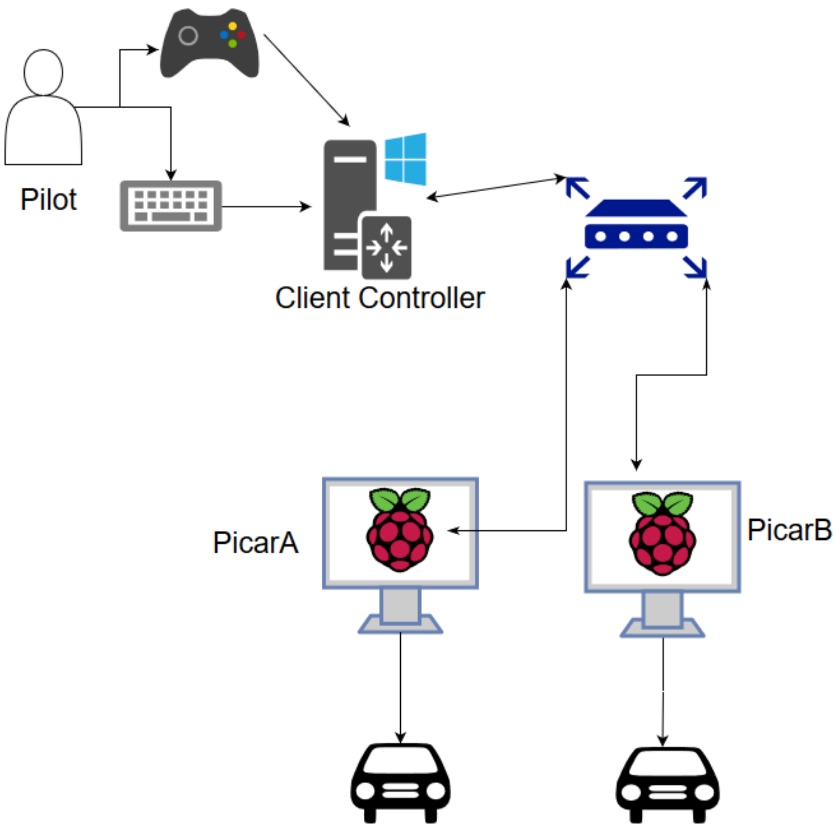


8 System Architecture and Design

8.1 Subsystem Identification

We have a clear separation between server and client subsystems. There must be a system which operates on the car hardware and controls vehicle motion, and there must be a system to send basic commands to the car remotely. Thus our system consists of a server which controls the hardware and a client which sends commands to the server. These are connected by standard IP, ethernet and WiFi technology.

8.2 Deployment Diagram



8.3 Persistent Data Storage

The main storage needed for the picars is simply the project git repository. As a dependency, there is also installing the Sunfounder Picar git repository. As part of this there exists a configuration file for the angle of the car wheels. The paths relevant to this file can be found in `src/setup/edit-alignment.sh`. Additionally, the camera calibrations are unique to each picar, meaning they are not preserved in git. By default the calibration is saved in `src/RobotServer/calibration.npz`. Care should be taken to preserve both of these files if peforming a reinstall of the repository.

On the client-side, there is a configuration file which is read from the user's documents folder, under the path `[Documents]/picar/gui_config.ini`. An example version of this file is found in `src/RobotClient/gui_config.ini`, and can be used as a template if copied to the above path.

For the Q-Learning algorithm that we ultimately implemented, we store the learned weights in a file `src/RobotServer/models/q.pkl`, which is reasonable since the file is small and we are storing essentially just a python dictionary and a few primitives.

8.4 Network Protocol

The previous team defined a gRPC network protocol to enable communication between the client controller and picars. We enhanced the protocol by altering the movement commands to the picar into a stream type to allow the picar to detect when the stream ends (e.g. by a connection timeout), allowing the picar to stop all movement in this case. In addition, we have made the server send back its protocol version with connection acknowledgement, allowing the client controller to detect when the server is using an unsupported version of the protocol, as opposed to giving cryptic error messages. We also added streaming the action taken by the follower with the video stream. Finally, we added a method to switch the follower model used by the picar in following mode. In all, the protocol includes a method

- for the client controller to switch between idle, leader, and follower mode
- to stream movement commands
- to obtain a stream which includes frames from the camera and actions
- to stop the stream
- to switch the follower model

8.5 Global Control Flow

The entry point for the picar code is `src/RobotServer/picar_driver.py`. This file initiates all the necessary SunFounder libraries, and begins with setting the picar in idle mode. Once here, the Client Controller is able to connect to the picar via the gRPC protocol, and thus it can be changed into a more useful mode. The picar enters a main loop which is essentially a switch statement depending on what mode it's in. In idle mode it does nothing and just waits to be switched into another mode. In leader mode it sends the movement commands it is given to the lowest level file in our code, `src/RobotServer/picar_helper.py`, which makes the calls to the SunFounder libraries to actually execute the desired movement. If instead the car is in follower mode the control flow becomes more interesting. The picar_driver file merely makes a call to its Follower object to get the desired movement command, but Follower (from `src/RobotServer/follower.py`) contains an instance of a Q-learner which it then queries to lookup the best action given its given state. The states, features, and reward functions are contained in the `follower.py` file. The actual extraction of the tag features is done in `src/RobotServer/tag_detection/detector.py`. This includes the pose estimation feature, which makes heavy use of the aruco library from OpenCV. Additionally, the follower can query the TurnController in `src/RobotServer/turn.py` to get its rule-based directional commands.

8.6 Hardware Requirements

The hardware requirements for the project are straightforward. The primary requirement is the SunFounder Picar-V kit [3], which contains all the physical car components, but notably, not a Raspberry Pi. The Raspberry Pi should be model 3B+ to ensure compatibility with the SunFounder kit and built in WiFi. Additional Raspberry Pi related necessities include a micro-usb power source (preferably 2 Amp) and microSD card (preferably 32GB).

A WiFi access point is also required so that the cars can be controlled while in motion. For best results, a dedicated WiFi router may be used. If initial Raspberry Pi network configuration is necessary, either HDMI and keyboard control or simply an ethernet cable will be required. It is strongly recommended to use an ethernet cable and determine the car's DHCP-negotiated IP address by checking router logs or similar, proceeding to login to the Pi over SSH (username: pi, password: capstone). This is preferred to using HDMI and configuring the network manually because the HDMI port will require an angle adapter to be reachable given the hardware setup of the car. However, also note that it is recommended to leave the network configured as DHCP as opposed to a static address, which will allow the car to be more easily connected to any future networks instead of hardcoding it to a single network.

The cars require a minimum of 4 Rechargeable UM-18650 3.7v Batteries, plus 2 for each additional PiCar. Of course, having more batteries available in reserve may make testing easier. Finally, a windows computer is required to run the controller client. For portability, it is probably easiest to use a laptop.

9 Algorithms and Data Structures

9.1 Algorithms

9.1.1 Tabular Q-Learning

The first reinforcement learning algorithm that we implemented was one-step tabular Q-Learning, and in fact this would be the algorithm we used in the final version albeit with several modifications to state and action from the first iteration. Each state-action pair has a value in the table representing the expected value of taking that action from that state. Policy is a simple argmax across all actions for a given state, so as to maximize expected value. Q-Learning is described by the following:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)]$$

In the above, S is a state, A is an action, R is the reward signal, α is the learning rate, γ is the discount factor, and $Q(S_t, A_t)$ is the tabular lookup of state-action pair S and A at time step t. In other words, the above equation says to change the value $Q(S_t, A_t)$ by the learning rate times the reward signal (the reward which appears at each step) plus the discount factor times the expected difference in the value of the previous and resulting state. Note that the value of a state-action pair takes into account its expectation of future rewards.

There is a rather significant drawback from using a one-step method: learning tends to be slower since it takes longer for the reward signals from further in the future to propagate back to older states. But, in return we get the significant advantage that off-policy training data can be used to train our model. This was largely non-negotiable for us, since training a weak model in our situation was a difficult problem (it would require manually resetting the cars and their positions over and over again!) So, with this approach we were able to use collected training data from a manually controlled follower and a leader that was following scripted commands, which was able to teach our model at least to the extent that the follower wouldn't immediately go off course and lose its way.

9.1.2 Actor-Critic Model

Actor-Critic combines the ideas of a policy gradient control with a value function. In essence, as opposed to policy being a simple argmax across a discrete set of actions, policy can itself be defined by a vector of weights. We used this to get a continuous output of throttle and direction, which our picars do support to an extent (based on the precision of the wheels and throttle hardware). This is achieved by having the policy model output a mean and standard deviation for both throttle and direction in state, and then randomly sampling the normal distributions to get the output. This is a rather fascinating idea for getting continuous output for the model, but the downfall of this model is we couldn't collect enough data to accommodate its more complex features and larger number of parameters. Also, while theoretically interesting, for our main use case of the leader-follower system, continuous throttle and directional output was not terribly important. This is because we mostly pilot the vehicles with maximum throttle anyway, and precise turning is a difficult enough problem that further increasing the possibilities for directional control is not terribly helpful.

9.1.3 Rule-based Turn Control

The primary problem with using these reinforcement learning algorithms is that they operate on every step, i.e. every time a the picar processes a new frame being received from its camera which is roughly every tenth of a second. This is fine for short-term decisions like increasing throttle to approach the leader car, but it becomes problematic when we consider a longer-term decision like initiating and executing a turn. Combining this with the fact that when the leader turns its tag often goes out of view for an extended number of frames, and it becomes clear that turning is a very hard problem for this model.

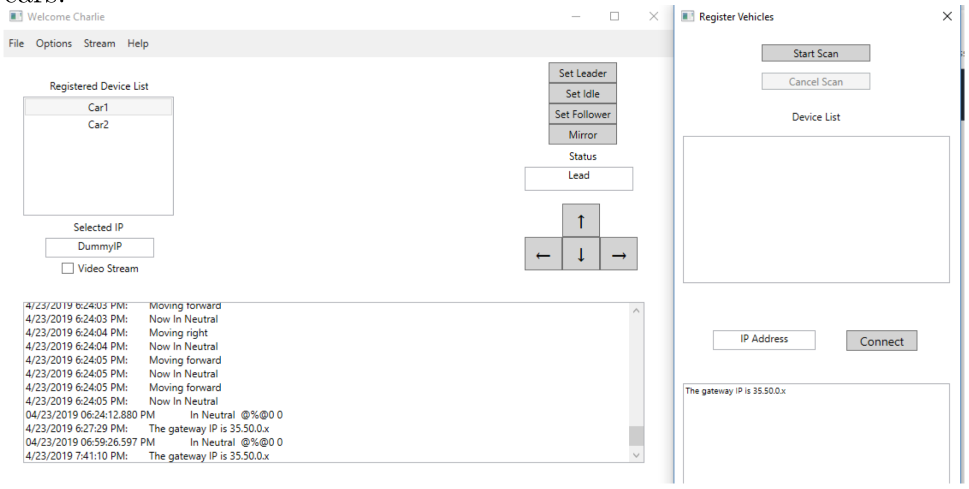
Our solution was to create a series of rules for when to initiate a turning procedure. Directional control would be left to this rule-based controller, while the throttle would continue to be controlled by the main reinforcement learning algorithm. With more time and data, such a system could later be made a reinforcement learning system, and we could consider the combined system a Hierarchical Reinforcement Learning system, which are on the developing fringe of machine learning at present.

9.2 Data Structures

The main data structure of interest for our algorithms was the table for tabular Q-Learning. From a low-level perspective, this is just a dictionary, but it should really be likened to a function with no generalization across states. The dictionary is indexed by the cross product of the tag state, which is the relevant features obtained by performing pose estimation using the leader's tag, and reversing state, which is meant to prevent the car from oscillating without good reason. The advantage of using a table to perform this mapping is that learning is theoretically easier because when learning, a step is taken to reduce loss in only that index of the table. This it is clear that the value function improves. The disadvantage, as compared to using a parameterized function approximation, is that there is no generalization between states which means learning is often slower in practice. With this in mind, the key to achieving good performance with limited training data is to reduce the number of states insofar as one reasonably can.

10 User Interface Design and Implementation

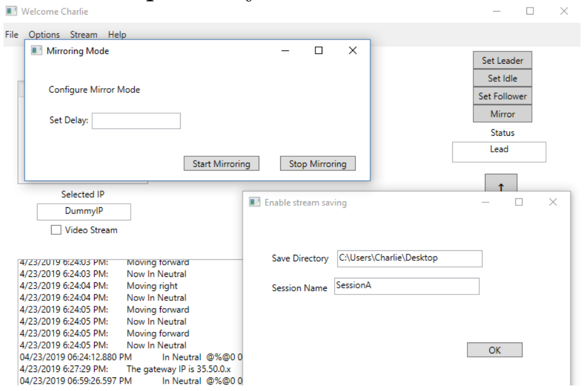
Our GUI was inherited from a previous team and we decided to keep this design while adding our own features. In the main window, the user is able to select a vehicle from the device menu and specify whether this vehicle is a leader, a follower, or idle. The user can then control the leader car with arrow buttons positioned in the bottom right-hand corner of the main window, or via the keybaord (WASD). There is a toggle button under the device menu that allows the user to decide whether or not they want the selected car's video stream to be enabled or not. If enabled, this stream is displayed in the center of the main window. At the bottom of the window, a log displays the movements of the leader car along with the corresponding time stamp. The file header allows the user to register vehicles, import a log, and export a log. The register window allows the user to scan for an ip address for a car or enter the ip address manually. Lastly, The options header gives the user the option of controlling the vehicle from the main window or controlling it with a controller. Below is a screenshot of the main window and the window for registering cars:



```
4/23/2019 6:24:03 PM:      Moving forward
4/23/2019 6:24:03 PM:      Now In Neutral
4/23/2019 6:24:04 PM:      Moving right
4/23/2019 6:24:04 PM:      Now In Neutral
4/23/2019 6:24:05 PM:      Moving forward
4/23/2019 6:24:05 PM:      Now In Neutral
4/23/2019 6:24:05 PM:      Moving forward
4/23/2019 6:24:05 PM:      Now In Neutral
04/23/2019 06:24:12.880 PM         In Neutral  @%@0 0
4/23/2019 6:27:29 PM:      The gateway IP is 35.50.0.x
04/23/2019 06:59:26.597 PM         In Neutral  @%@0 0
4/23/2019 7:41:10 PM:      The gateway IP is 35.50.0.x
```

```
The gateway IP is 35.50.0.x
```

We added a replay functionality to the GUI that enables the the user to control the leader while the replaying car receives commands on a delay. In order to send replays, the user simply selects the leader from the device menu then presses the mirroring button in the main window. Another window then pops up which allows specification of the delay. Another functionality that we added is giving the user the option of saving the stream from the follower by enabling the stream saving header. The stream images are sent to a csv file with a location specified by the user. We also added a functionality such that when the GUI loads it automatically connects to the cars and sets each car to leader, follower, or idle. It also loads a log automatically. This is all based on a .ini file that the user specifies. This adds the convenience of not having to register vehicles repeatedly. Below is a screenshot of the mirroring and stream saving window:



Save Directory: C:\Users\Charlie\Desktop

Session Name: SessionA

```
4/23/2019 6:24:03 PM:      Moving forward
4/23/2019 6:24:03 PM:      Now In Neutral
4/23/2019 6:24:04 PM:      Moving right
4/23/2019 6:24:04 PM:      Now In Neutral
4/23/2019 6:24:05 PM:      Moving forward
4/23/2019 6:24:05 PM:      Now In Neutral
4/23/2019 6:24:05 PM:      Moving forward
4/23/2019 6:24:05 PM:      Now In Neutral
04/23/2019 06:24:12.880 PM         In Neutral  @%@0 0
4/23/2019 6:27:29 PM:      The gateway IP is 35.50.0.x
04/23/2019 06:59:26.597 PM         In Neutral  @%@0 0
```

11 Testing

For our project, leveraging a standard unit-testing scheme was difficult. This is primarily because most code is dependent on being run on the picar hardware in order to operate normally. It would be unmanageable to create unit tests for simple hardware functions because of the relatively high effort involved in starting up the picar system every time.

Another issue with testing is that from the point of view of performance of a model it is difficult to have a true metric of how well the follower is performing. This is exaggerated because camera shake and sharp movements cause the tag to be unidentifiable in many frames in which it should otherwise be visible. Further, because of the low battery life of the picars and the fact that they ordinarily function on their own dedicated network, using a framework for testing would have been difficult.

Considering this, our scheme for testing was to design the low-level algorithmic layers in such a way that they did not have the picar hardware as dependencies. We were then able to create small tests in each of the algorithmic files that could possibly be run either with the picar hardware or without it (depending on the nature of the test). Then, to run the test usually involved running the given python file as a main script.

12 Project Management

12.1 Project Plan

Sprint 1: Acquire vehicles, confirm working status, and begin becoming familiar with existing codebase.

Sprint 2: Allow controller machine to record image data it has sent; begin mirroring mode implementation; add ability to control camera movement.

Sprint 3: Update server code to Python 3.5; finish Mirroring mode; collect training data; train and test supervised machine learning model

Sprint 4: Adjust model hyperparameters; begin work on reinforcement learning model

Sprint 5: Extract more features from tag; implement Actor-Critic reinforcement learning model

Sprint 6: Adjust tabular model to accommodate new features

Sprint 7: Small optimizations and rule-based turning control

Sprint 8: Refine rule-based turning control and following distance

12.2 Risk management

As a research-focused project, we did not have to be as risk-averse as we ordinarily would. Our goal was to be exploratory with regards to the algorithms and techniques for the leader-follower system, but the primary risk was in our machine learning approaches in general not working with the data we were able to collect. In this case we would fallback to a purely rule-based system. We did end up going back to a rule-based system for turning behavior, since getting sufficient data to teach the system to turn proved very difficult, but we were able to successfully use a reinforcement learning agent for throttle control.

References

[1] R. S. Sutton and A. G. Barto, Reinforcement learning: An Introduction, 2nd ed. Cambridge, Massachusetts: The MIT Press, 2018.

[2] OpenCV Documentation `https://docs.opencv.org/3.1.0/d5/dae/tutorial_aruco_detection.html`

[3] Sunfounder PiCar-V kit `https://www.sunfounder.com/smart-video-car-kit-v2-0.html`